Skriptaus arkkitehdin metodina

Diplomityö
Oulun yliopisto, Arkkitehtuurin tiedekunta
Maija Poukka
Pääohjaaja Rainer Mahlamäki
Pvm 07.02.2018

TIIVISTELMÄ

Skripti tarkoittaa tietotekniikassa komentosarjaa tai ohjelmaa, joka kirjoitetaan skriptikielellä. Skriptaukseen viitataan usein ohjelmointina. Diplomityössäni olen kiinnostunut skriptauksesta metodina tutkia arkkitehtuurisuunnittelua. Sen mahdollisuuksista luoda arkkitehtuurin muotokieltä sekä kehittää arkkitehdin esitys- ja työskentelytapoja. Diplomityössäni tutkin arkkitehtuurin muotoilua ja geometrian luomista algoritmiavusteisesti Python-skriptauksen keinoin.

Diplomityöni on tutkimuspainotteinen, mutta sisältää myös suunnittelutyön. Diplomityöllä minun on tarkoitus syventyä skriptaukseen arkkitehtuurin metodina: minkälaisia ominaisuuksia skriptauksella on arkkitehdin näkökulmasta ja miten se vaikuttaa arkkitehtuurin muotoiluun. Suunnittelen diplomityössäni arkkitehtuurin muotoiluun kolme erilaista skriptaamalla rakennettua työkalua. Työkalut muodostavat tietynlaista geometriaa ja näin ollen työkalut ovat samaan aikaan muotoilututkielmia. Kutsun työkaluja diplomityössä tutkielmiksi. Diplomityöni esittelee prosessin tutkielmien luonnin taustalla.

Diplomityöni jakautuu kahteen osaan. Ensimmäinen osa esittelee diplomityön kontekstin ja motiivin skriptata arkkitehtuurin työkaluja. Se koostuu esseistä, jotka käsittelevät ohjelmointia ja tietokoneiden käyttöä arkkitehtuurissa 1940-luvulta lähtien tähän päivään. Teksteissä otan esille joitakin arkkitehtuurin suuntauksia, jotka ovat syntyneet tai yleistyneet tietokoneiden käytön myötä. Esseet ovat sävyltään pohdiskelevia. Toinen osa kuvaa diplomityön prosessia. Toisessa osassa esittelen muotoilututkielmat. Jokainen tutkielma sisältää kolme rendausta mallista, skriptin ja skriptin selostuksen. Tutkielmissa käsittelen skriptauksella saavutettavia ominaisuuksia kuten attraktori, rekursio ja satunnaisuus. Hyödynnän skripteissä matemaattisia operaattoreita. Diplomityöhöni kuuluu lisäksi tutkielmia esittelevät planssit. Diplomityön liitteenä on perehdytys Python-ohjelmoinnin perusteisiin. Kokoan liitteessä yhteen Python-kielen ohjausrakenteita, joita käytän diplomityön toisen osan muotoilututkielmien rakentamisessa.

Suunnittelutyön lisäksi diplomityö on minun puheenvuoroni ja pohdintani vuonna 2017-2018 ajankohtaiseen aiheeseen: ohjelmointi työkaluna arkkitehtuurissa. Diplomityöni alkaa tekstillä ”Miksi arkkitehdin tulee osata ohjelmointia?” ja päättyy omaan reflektiooni aiheesta.

ABSTRACT

A script means in computer science an execution of tasks that is written in a scripting language. Scripting is often referred as programming. In this thesis I am interested in scripting as a method to study architecture. I am interested in its possibilities to explore architecture form and to develop architect's presentation and working methods. I practice algorithmic aided architectural design and geometry through scripting methods.

This thesis of mine is research oriented work including also design work. I will study scripting as an architectural method: what kind of qualities does it have from architect's point of view and how does it affect architectural form. I design three architectural design tools through scripting. The tools form specific geometry and this way while being as tools, they work as well as design studies. The tools are called as studies in this thesis. The thesis presents process behind the studies.

My thesis is divided into two chapters. The first chapter introduces the context of the thesis. It consists of essays which describe programming and the use of computers in architecture from the 1940s till nowadays. In essays I describe some architectural tendencies that have arisen by use of computers. These essays have speculative tune and they reflect the tendencies to trends that we see in architecture today. The second chapter presents the design process of the thesis. In the second chapter I present the design studies. Every study includes three rendering, a script and comments of the script. In studies I show some features that coder could achieve by using scripting as an example attractor, recursion, random and use mathematical operations. Each study concludes a presentation board. In the appendix of the thesis I introduce a short overview of Python programming basics. In the appendix I collect Python-language structures which I use for the design studies in the second chapter.

In addition this thesis is a platform of mine to make a statement about currently the hot topic of the year 2017-2018 as programming as a tool in architecture. In this thesis I contemplate, if architect or architecture student should or shouldn't learn how to program. My thesis begins with text" Why architect should learn to programming?" and ends with a critical reflection about the topic.

Oulun yliopisto, Arkkitehtuurin tiedekunta

Tekijä

Maija Poukka

Pääohjaaja

Rainer Mahlamäki

Ohjaajat

Tuulikki Tanska, Toni Österlund

Työn nimi

Skriptaus arkkitehdin metodina

Työn sivumäärä

80

Muut

3 planssia

Koulutusohjelma

Arkkitehdin tutkinto, arkkitehtuurin koulutusohjelma

Opintosuunta

Nykyaikainen arkkitehtuuri

Oulu, 07.02.2018

Maija Poukka

SISÄLLYSLUETTELO

1. JOHDANTO

Skripti tarkoittaa komentosarjaa, jolla johdetaan tietokoneen ohjelmaa halutun toiminnan toteuttamiseksi. Skripti koostuu joko yhdesta tai useammasta algoritmista. Algoritmi tarkoittaa tiettyä toimintaohjetta tai toimintaohjejoukkoa, joka ratkaisee tietyn ongelman annettujen lähtötietojen avulla.

Tässä diplomityössä skriptillä viitataan lausekieliseen tekstimuotoiseen ohjelmaan, jota kirjoitetaan tekstitiedostoon. Lausekielessä ohjelman kieli muistuttaa luonnollista kieltä. Kieli koostuu tunnuksista eli nimistä ja varatuista sanoista, joita käytetään kielen ohjausrakenteisiin. Tietokone kääntää lausekieltä bittitasoiseksi konekieleksi. Skriptaukseen perustuva mallintaminen hyödyntää ohjelmointia mallintamisympäristössä.

Termit ohjelmointi ja skripti ovat väljät ja niitä käytetään usein sekaisin. Sanat voidaan kuitenkin myös jaotella seuraavasti: Skriptillä voidaan viitata alemman tason ohjelmointiin ja ohjelmointi sanalla viitata korkeamman tason toimintaohjeisiin. Tällöin skripti toimii ylemmän tason ohjeilla. Käytän tässä diplomityössä termiä ohjelmointi skriptauksen yläkäsitteenä.

Vuonna 2011 aloittaessani arkkitehtuurin opinnot Oulussa DigiWoodLab:n aktiivinen toiminta arkkitehtuurin osastolla oli päättymäisillään Oulussa sen aloittajien mukana. DigiWoodLab-projektin tarkoituksena oli tutkia uusia tuotantomenetelmiä puurakentamisessa. Projektissa hyödynnettiin algoritmiavusteista suunnittelua. En itse päässyt osallistumaan algoritmiavusteisin suunnittelun opetukseen ja koin koulutukseni tältä osin puutteelliseksi. Tällä diplomityöllä syvennyn tähän aiheeseen.

Tämä työ on lähtenyt liikkeellä siis omasta kiinnostuksesta skriptaukseen ja algoritmiavusteiseen suunnitteluun. Ohjelmointi on ollut esillä paljon mediassa vuonna 2017 ja myös sitä ennen: Pelifirmat ovat saaneet Suomessa laajasti näkyvyyttä. Alakoulun opetussuunnitelmaan on lisätty ohjelmointi. Linda Liukas ja Henrietta Kekäläinen ovat kannustaneet nuoria ohjelmoinnin ja teknologian pariin. Mielestäni aiheeni skriptaus arkkitehtuurin metodina on ajankohtainen.

Tämän diplomityön aiheeseen perehtymisen ja harjoittelun olen aloittanut lähes tyhjältä pöydältä. Skriptauksesta minulla on aikaisempaa kokemusta ainoastaan html–kielellä kirjoitetuista omista internet-kotisivuista. Osana tätä diplomityötä olen opetellut myös Grasshopperin käyttöä.

Avainkäsitteitä:

Algoritmi

tarkoittaa tietokoneella luotua äärellistä joukko ohjeita tai käskyjä, jotka ohjaavat prosessia. Algoritmi ratkaisee jonkun ongelman sille annetuista lähtötiedoista. Algoritmeja voidaan havainnollistaa Turingin koneen avulla.

Komputaatio

laskenta, englanniksi computation. Komputaatio tarkoittaa jonkin määrätyn matemaattisen prosessin seuraamista. Komputaatioon liittyy usein ajatus tuntemattoman ja epämääräisen tutkimista, johon ihmismieli ei yksin pystyisi. Komputaatiolla voidaan kasvattaa ihmisen älyllisiä ulottuvuuksia tietokoneen avulla.

Metodi

menetelmä, viittaa tapaan, jolla tutkimusaineistoa kerätään tai analysoidaan.

Ohjelmointi

tarkoittaa toimintaohjeiden eli algoritmien tekemistä. Esimerkiksi algoritmeja voidaan kirjoittaa tietokoneella ohjelmointikielen avulla. Ohjelmointi voi tarkoittaa yleensä ylemmän tason vaativampia komentoja, mutta sillä voidaan myös viitata alempaa komentosarjaan (ks. skriptikieli). Termi on varsin väljä.

Ohjelma

ohjelma on joukko ennalta laadittuja käskyjä, joilla tietokone suorittaa sille laaditun tehtävän. Skripti voi toimia ohjelmana.

Ohjelmisto

ohjelmisto on useista ohjelmista koostuva kokonaisuus.

Parametrinen

suunnittelussa keino mallintaa assosiatiivista geometriaa. Parametrisessa suunnittelussa objekteilla on ominaisuuksia eli parametreja, joita pystyy nopeasti muokkaamaan halutun kaltaiseksi parametria muuttamalla. Assosiatiivisuus eli liitännäisyys tarkoittaa matematiikassa, että (a•b)• c=a• (b• c) on totta, kaikilla a, b, ja c arvoilla. Näin ollen parametrialla ei yleensä ole rekursiivisia (ks. rekursio s.70) ominaisuuksia. (David Jason Gerber, 2014)

Skriptikieli

komentosarja, jolla useimmiten viitataan alemman tason ohjelmointiin, jolla voidaan automatisoida tietokoneen toimintoja. Skriptejä voi esimerkiksi syöttää suoraan tietokoneohjelmistoille ohjelmistolle sopivalla kielellä. Python on esimerkiksi skriptikielestä.

Turingin kone

Turingin koneen avulla selitetään usein ohjelmoitavan tietokoneen toimintaa. Se on teoreettinen malli algoritmien havainnollistamiseksi.

2. KIRJALLISUUSOSA

2.1 MIKSI ARKKITEHDIN TULEE OSATA OHJELMOIDA?

Se on kieli (ohjelmointi), jonka nuoren ihmisen on osattava (Bengt Holmström, 2017).

Filosofi Ludwig Wittgenstein pohti aikoinaan kielen merkitystä ajatteluumme. Kuinka kieli, jota käytämme luo raamit myös ymmärryksellemme ja mahdollisesti kaventaa näkemystämme tästä olevasta. Samankaltaisesti voimme myös ajatella arkkitehtuurissa CAD-ohjelmien vaikutuksesta arkkitehtuurisuunnitteluun: Suunnitteluohjelmistot ovat suuri apu arkkitehdin arkipäivässä, mutta niihin rajoittautuminen estää uuden ja innovatiivisen arkkitehtuurin luomisprosessia. Ohjelmistot tarjoavat suhteellisen standardivalikoiman työkaluja, joiden puitteissa arkkitehdin tulee toimia. Täten ohjelmistot tuottavat usein hyvin samankaltaista arkkitehtuuria: suorat seinät, standardisoidut ikkunakomponentit, ovenkahvat jne. Ohjelmistot kaventavat siis uusien suunnitteluratkaisujen syntymistä, koska arkkitehti ei ole täysin vapaa ohjelmiston asettamista rajoitteista.

Arkkitehdit ovat paljolti ohjelmistokehittäjien varassa minkälaista arkkitehtuuria ohjelmistokehittäjät ovat kykeneviä luomaan. Mutta mitä, jos arkkitehti pystyisi itse ottamaan ohjelmistokehittäjän paikan: luomaan omia työkaluja tai muokkaamaan niitä. Ohjelmoimaan itse tietokoneen luomaan uutta geometriaa.

Arkkitehdin työ on siirtynyt parissa kymmenessä vuodessa piirustuspöydiltä tietokoneille. Tietomallinnus tehdään Revitillä tai Archicadillä. Kuvat rendataan ja malli tulostetaan 3d-printterillä. Luonnostelu edelleen syntyy osaksi paperilla, mutta pääosin koko suunnittelu tehdään tietokoneella. Tietokone toimii arkkitehdin kynänä. Sillä piirretään samoja viivoja, mitä vuosisadat on piirretty jo piirustuspöydillä. Tietokone mahdollistaisi arkkitehdin tehdä jotain uutta, jos sen koko potentiaali otettaisiin käyttöön.

Ben Fry (Ph.D, Media Laboratory, MIT) ja Casey Reas (prof. UCLA Department of Design/Media Art) ovat lausuneet, että on vain ajan kysymys, milloin ohjelmoija ja arkkitehti on sama henkilö, ja milloin koodi korvaa piirustuksen (Mike Silver, 2009). Fry ja Reas ovat olleet uranuurtajia tuodessaan suunnittelijoita ja taitelijoita ohjelmoinnin pariin. He ovat kehittäneet MIT:lla (the Massachusetts Institute of Technology) Processing-kielen visuaalisten alojen suunnittelijoiden käyttöön. Myös Mark Burry (arkkitehti, prof.) ohjelmointia käsittelevässä kirjassaan ”Scripting Cultures” toteaa, että arkkitehtuurin perusfundamentit, kynä ja paperi on jo haastettu ja kehitys tulee muuttamaan alaa (Mark Burry, 2011). Uudet digitaaliset menetelmät luovat uusia mahdollisuuksia tuotantoon ja suunnittelun ja tuotannon välinen yhteys tiivistyy. Suunnittelupöydältä voidaan tulevaisuudessa suoraan ajaa suunnitelmia toteutukseen ilman välikappaleita (Lisa Iwamoto, 2009). Arkkitehtuurissa piirustus, perinteisenä arkkitehtuurin välineenä, menettää merkitystänsä rakennusprosessissa (Robin Evans, 1997).

Samoin Suomessa ohjelmointi kasvattaa jatkuvasti suosiotaan. Ohjelmointi on nykyisin pakollinen osa peruskoulun opetussuunnitelmaa, ja monet pelifirmat ja sovellusyritykset ovat niittäneet mainetta Suomessa ja Suomen ulkopuolella. Tulevaisuudessa kenties teknilliseen yliopistoon tai arkkitehtuurin laitokselle hakautuvien opiskelijoiden odotetaan osaavan perustaidot ohjelmoinnista jo ennen yliopistoon tuloa.

Ohjelmoinnista on tulossa keskeinen osa yleissivistystä ja sen sanotaan siivittävän neljänteen teolliseen vallankumoukseen. Neljännen teollisen vallankumouksen ytimessä ovat digitalisaatio, automatisointi, robotiikka, tekoäly ja 3d-tulostus, joiden kaikkien edellytys ohjelmointi on. Arkkitehtuuri tulee myös muuntumaan näiden myötä. Robotiikka ja 3d-tulostus mahdollistavat uudet tuotantomenetelmät rakennusalalla. Digitalisaatio yhdistää rakennusalan tekijöitä. Algoritmien hyödyntämiseen perustuvasta arkkitehtuurista tulee valtavirran arkkitehtuurin suuntaus eikä vain marginaalin puuhastelua.

Ohjelmointi tekee arkkitehtuurista entistä monimutkaisempaa, koska tietokoneiden mahdollisuudet ovat laajat: Tietokone säilöö ja toistaa suuria määriä dataa. Se laskee uusia vaihtoehtoja ja iteroi ratkaisuja sekunneissa. Se pystyy muotoilemaan mitä monimutkaisempaa algoritmista geometriaa, jota tuskin pystyy edes kuvittelemaan. Myönnettäköön, että tänä päivänä Grasshopper ja Dynamo ovat jo tuoneet algoritmit lähemmäs arkkitehtejä, mutta kyseisissä ohjelmissa tulee niiden rajat lopulta vastaan. Osa rajoista voi olla ylitettävissä ohjelmoinnin keinoin. Esimerkiksi rekursiivisuuden (ks.rekursio) luominen tietokoneen laskemana on mahdollista vain ohjelmoimalla.

Ohjelmointi säästää aikaa, ja manuaalista työtä poistuu, kun toistuville työvaiheille voidaan luoda koodi. Työskentelystä tulee näin entistä tehokkaampaa, ja tehokkuus vapauttaa aikaa toisille työtehtäville, kuten suunnittelulle. Esimerkiksi kyse voi olla tietyistä toiminnoista, jotka tulee suorittaa useille tiedostoille kerralla. Koodi vain syötetään tietokoneelle ja jätetään tietokone suorittamaan itsenäisesti toimenpiteet. Koodeja voidaan edelleen tallettaa ja käyttää uudelleen, kun sama tilanne tulee uudemman kerran vastaan. Ylipäätänsä ohjelmoinnilla voidaan vähentää siis kaikenlaista tietokoneella tehtävää työtä, johon liittyy toistuvuutta.

Ohjelmoivilla arkkitehdeillä voi olla keskeinen rooli edessä olevassa arkkitehtuurin murroksessa. Ohjelmoivat arkkitehdit lisäävät esimerkiksi painetta CAD-ohjelmien tuotekehitykseen. On tärkeää, että arkkitehdit ymmärtävät suunnittelutyökalujen toimintaa. Monet merkittävät alan suuntaukset nivoutuvat jollain tavoin kehittyneeseen tietokoneen hallintaan. Esimerkiksi arkkitehtuuri näyttää tällä hetkellä suuntaavan kokonaisvaltaiseen systeemiajatteluun, jossa arkkitehtuuri, materiaalit, ympäristö ja talotekniikka muodostavat yhden yhtenäisen yksikön. Systeemiajattelu voidaan toteuttaa ohjelmoimalla. Ohjelmoimalla toteutetussa systeemiajattelussa ensin kartoitetaan eri tekijöiden raja-arvot ja optimit. Lopuksi generoidaan ohjelmalla paras mahdollinen suunnitelman lopputulos raja-arvojen ja optimien vaihtoehtopilvestä.

Kaiken lisäksi ohjelmointi on hauskaa eikä ollenkaan niin vaikeaa, kuten moni ehkä luulee. Se kehittää arkkitehdin algoritmista ajattelua sekä ymmärrystä tietokoneiden ja ohjelmien toiminnasta. Se on palkitseva harrastus, kun aikasi tapeltua koodin kanssa löydät siitä vihdoin virheen, ja tietokone lopulta lukee koodin onnistuneesti.

2.2 OHJELMOINTI ARKKITEHTUURISSA

Skriptaus on vaikuttanut arkkitehtuurin praktiikkaan, tyyliin, esitystapoihin ja teoriaan. Kirjallisuuskatsauksen tässä kappaleessa kartoitan motiiviani tehdä arkkitehtuurin muotoiluun työkaluja skriptaamalla. Yritän löytää skriptauksen erityispiirteitä ja sen luonnetta arkkitehdin metodina arkkitehtuurin muotoilussa. Skriptaamalla luotavia työkaluja voidaan levittää tai hyödyntää ohjelmiston sisällä toimivina työkaluina, ohjelmiin liitettävinä plugineina tai ohjelmien välillä operoivina liitoksina. Skriptaamalla ja/tai ohjelmoimalla voidaan rakentaa kokonaisina ohjelmistoja. Skriptaamalla luotu työkalu voi olla generaattori, jolloin sillä voidaan ratkaista arkkitehtonisia ongelmia, tai jonkin tietyn tehtävän suorittava operaattori.

Komputationaalinen arkkitehtuuri on laaja aihe ja sisältää useita eri suuntauksia. Seuraavat esseet tuskin antavat kattavaa kuvaa skriptauksesta metodina arkkitehtuurissa. Esseiden haasteena on löytää rajanveto skriptaustyökalujen ja ohjelmistojen välille. Nykymuotoinen ohjelmoitu tietokone ja sen ohjelmistot toimivat tekstimuotoisen ohjelmoinnin tai skriptauksen keinoin, jolloin kaikki tietokoneen avulla työstetty arkkitehtuurin voidaan katsoa ohjelmoinnin tai skriptauksen piiriin. Kysymys on kuitenkin siitä, milloin skriptauskulttuuri tai nimenomaan skriptaukseen liittyvät ominaispiirteet ovat vaikuttaneet arkkitehtuuriin ilmentymiseen siten, miltä se tällä hetkellä näyttää.

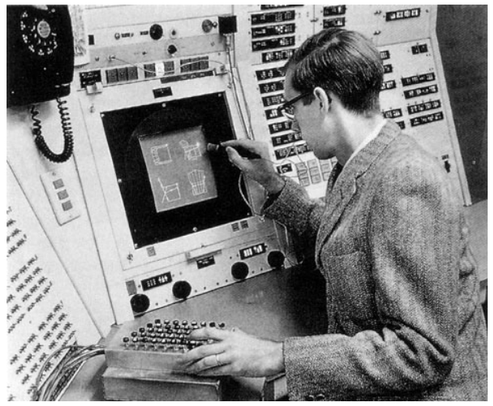
Tätä katsausta kirjottaessani olen erityisesti lukenut Mark Burrya, John Frazeria, Christopher Alexanderia, William J. Mitchelliä, Kostas Trezidisiä, Paul Coatesia Casey Reasia sekä Patrik Schumacheria. Muita, joiden teksteihin kannattaa perehtyä, ovat esimerkiksi Ulrich Flemming, Chris Yessios, Tom Mather, Alan Bridges, Chuck Eastman, George Stiny, Terry Knight, John Gero, Ranulph Glanville, Phil Steadman, Lionel March ja John Maeda. Lisäksi kannattaa tutustua järjestöihin, kuten Siggraph, Smartgeometry, eCAADE, ACADIA ja CAADRIA.

2.2.1 CAD-ohjelmien kehitys

CAD (Computer-Aided-Design) eli tietokoneavusteinen suunnittelu viittaa laajaan valikoimaan software-ohjelmistoja, joita insinöörit, arkkitehdit ja designerit käyttävät suunnittelutöissään. CAD-ohjelmistoilla voidaan suunnitella esimerkiksi rakennuksia, siltoja, autoja ja eri kulutustuotteita.

Ohjelmoinnin historia on yhtä pitkä kuin ohjelmoitavien nykymuotoisten tietokoneidenkin eli yli seitsemän vuosikymmentä. Alkuun kaikki ohjelmointi on ollut bittisarjaa, josta vähitellen eri ohjelmointikielet ja skriptaus ovat kehittyneet. Arkkitehtien hyödyntämien CAD-ohjelmisto-

jen, kuten muidenkin software-ohjelmistojen toiminta, pohjautuu ohjelmointiin tai skriptaukseen. CAD-menetelmät lähtivät kehittymään toisen maailman sodan jälkeen ohjelmoitavien tietokoneiden kehityksen myötä. 1950-luvulle tultaessa MIT:ssa työstettiin numeraaliseen ohjaukseen perustuvia laitteita ja pyrittiin automatisoimaan insinöörien suunnittelua. Ensimmäiset graafiset suunnitteluvälineet kehitettiin 1960-luvulla, kuten esimerkiksi Sketchpad, GRASP ja LOKAT.



Kuva 1
Ivan Sutherland ja Sketchpad vuodelta 1962.

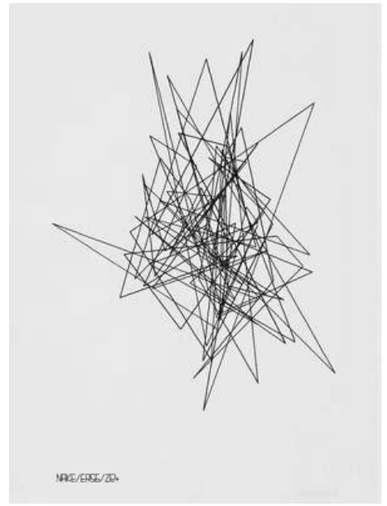
Sketchpadia (kuva 1.), joka oli yksi ensimmäisistä CAD-ohjelmistoista, voidaan pitää erityisen vallankumouksellisena. 1960-luvun alussa Ivan Sutherlandin väitöskirjanaan MIT:lle kehittämä Sketchpad pystyi jo piirtämään, siirtämään sekä muuttamaan objekteja. Sketchpad esitteli esimerkillisesti CAD-ohjelmistojen keskeisen käsitteen: parametri. Sketchpadissa kappaleilla oli parametreja, joita muuntelemalla objektien tai kappaleiden välisiä suhteita pystyi joustavasti muuttamaan. Sean Ahlquist ja Achim Menges kirjoittavat ”Computional Design Thinking”-julkaisun johdannossa, että Sketchpad loi ajatuksen, jossa symbolisesti design-suunnittelu ei enää esittänyt fyysisiä elementtejä, vaan ennemmin summan voimia ja rajaehtoja, joista muoto rakentui. Ensimmäiset kokeilut olivat tyypillisesti uniikkeja. CAD-ohjelmien saavuttaessa kuitenkin suuremman yleisön Sketchpadin ja muiden ensimmäisten CAD-ohjelmien metodit ja objektit standardisoituvat. (Sean Ahlquis, Achim Menges, 2011)

1970-luvulla 2d-piirtämisestä siirryttiin tutkimaan 3d-mallinnusta. Ken Versprillen väitöskirjassaan kehittämä NURBS mahdollisti perusteet 3d-kaarille ja -pinnoille ja Alan Grayerin, Charles Langin ja Ian Braidin kehittämä PADL (Part and Assembly Description Language) mahdollisti kiinteiden kappaleiden mallinnuksen tietokoneella. 1980-luvulta lähtien arkkitehdit ovat hyödyntäneet CAD-ohjelmia enenemissä määrin toimistoissaan CAD-ohjelmistojen kaupallistuttua 1980-luvulla. Ensimmäinen Autocad-versio julkaistiin vuonna 1983. (David Cohn, 2010)

Mark Burry kirjoittaa vuonna 2011, että skriptauskulttuuri arkkitehtuurissa on syntynyt tietynlaiseksi vastakulttuuriksi konventionaalisille CAD-ohjelmistoille. CAD-ohjelmistojen heikot mahdollisuudet luovaan ja monimuotoiseen suunnitteluun on aiheuttanut tyytymättömyyttä arkkitehtien parissa. Digital Project, Generative Components, Processing ja Grasshopper Rhinoon ovat osakseen olleet paikkaamassa tätä tyytymättömyyttä. Tarvetta on vapaampaan työskentelyyn kannustavalle käyttöliittymälle, jonka avulla voidaan esimerkiksi integroida sujuvammin useampia ohjelmistoja yhteen, ja työkaluille, jotka ovat vähemmän riippuvaisia alustastaan. (Mark Burry, 2011)

2.2.2 Architectural Design-lehtikatsaus ja ohjelmoinnin vaikutuksista arkkitehtuurin muotoon

Ohjelmointi tai skriptaus ja skriptauskulttuuri ovat vaikuttaneet arkkitehtuuri- ja taidesuuntauksiin. Skriptaamalla geometriaa voidaan rakentaa matemaattisilla operaattoreilla. Matemaattiset operaattorit mahdollistavat generatiiviset ominaisuudet ja yhtälöihin perustuvan geometrian (Andrew Saunders, 2013). Skriptaus on tuonut uusia työkaluja arkkitehdeille ja taiteilijoille. Casey Reas (prof. UCLA Department of Design/Media Art), toinen Processing-kielen kehittäjistä, pitää skriptausta taidemuotona. Casey Reas toteaa vuonna 2006 kirjoitetussa ”Architectural Design”-lehden artikkelissa, että skirptaus ja sovellukset ovat hänen välineensä tehdä taidetta (Casey Reas, 2006).



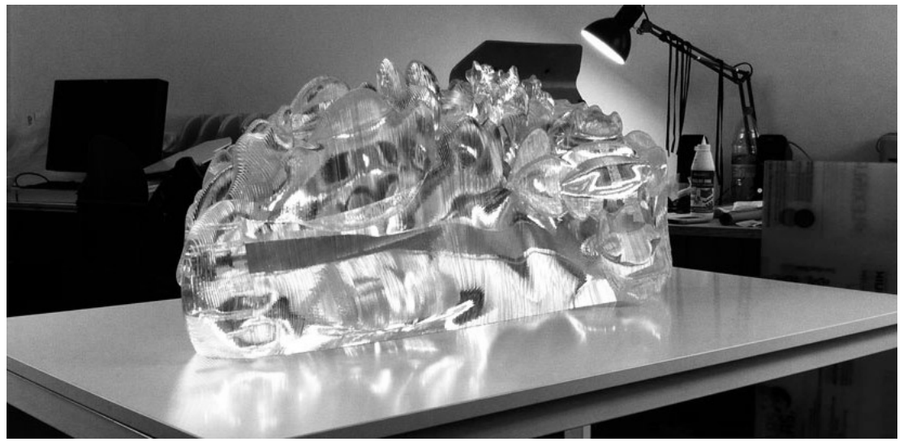
Kuva 2
Frieder Nake, 6/7/64 Nr. 20 Zufälliger Polygonzug, 1963.
Frieder Naken teoksessa vuodelta 1963 algoritmi valitsee ensin satunnaispisteet piirtoalueelle ja yhdistää pisteet suorilla viivoilla. (Frieder Nake, 2017)

Algoritmeihin perustuvaa taidetta ja arkkitehtuuria on luotu ainakin 1960-luvulta lähtien. Tutkijat ja insinöörit tuottivat ensimmäiset generoidut kuvat tietokoneella 1950-luvulla (Casey Reas, 2006), ja 1960-luvulla lähtien generatiivinen taide on ollut jo suuntaus. Tunnettuja generatiivisen taiteen tekijöitä 1960-luvulta ovat olleet esimerkiksi Frieder Naken, Hiroshi Kawano ja Kerry Strand. He itse tai tutkijan avustamana ohjelmoivat tietokoneen piirtämään esimerkiksi viivaa tai värikenttiä (kuva 2.). Arkkitehtuurissa John Frazeria on pidetty yhtenä pioneerina tuodessaan tietokoneavusteisen suunnittelun arkkitehtuuriin. Hän on tietävästi ensimmäinen arkkitehtiopiskelija, joka piirsi lopputyönsä tietokoneen avulla (Mark Burry, 2011). Frazer kehitti myös arkkitehtuurin teoriaa ja toi generatiivista suunnitteluideologiaa arkkitehtuuriin ”An Evolutionary Architecture”-tutkimuksellaan.

1980-luvulta lopulta voidaan alkaa hahmottaa modernin arkkitehtuurin kurinalaisen niukan muodon alkavan hajota 2000-luvulle aina jopa epärationaaliseksi ornamentiikaksi. Tietokoneen ja algoritmiavusteisen suunnittelun rooli kehityksessä on keskeinen, sillä sen avulla arkkitehtuurissa pystytään helpommin hallitsemaan monimutkaisempia ja toistoa suosivaa geometriaa. Skriptaus on yksi algorimiavusteisen suunnittelun metodi. Yhtenä mahdollisena käännekohtana voidaan ehdottaa vuonna 1988 Museum of Modern Art:n (MOMA) tiloissa (Michael Meredith, 2013), New Yorkissa, järjestettyä näyttelyä ”Deconstructivist Architecture”. Sen järjestivät mm. Rem Koolhaas, Zaha Hadid, Eisenman, Coop Himmelblau, Frank Gehry, Daniel Libeskind ja Bernard Tschumi. Näyttelyssä keskiössä oli arkkitehtoniset objektit ja esille nousi tietynlaiset arkkitehtuurin häiritsevät ominaisuudet. ”Deconstructivist Architecture” oli näyttely, jossa rikottiin stabiilia, tasapainoista ja klassista hierarkiaa. Työt olivat usein levottomia, dynaamisia ja epävakaita (Akos Moravanszky, Torsten Lange, 2017). Näyttelyn järjestäjät on myöhemmin yhdistetty algoritmiavusteiseen suunnitteluun.

Vuonna 1997 Gregg Lynn ehdottaa ”Architectural Design”-lehden ”Architecture After Geometry”-numeron artikkelissa ”Adcanced Form of Movement” arkkitehtuuriin ”motion-form” eli liikkuvaa muotoa. Hänen mukaansa ainoa tapa rakentamisessa ottaa huomioon virrat ja voimat, jotka jatkuvasti muokkaavat rakennusta ja ympäristöä, on työntää arkkitehtuuri liikkeeseen. Tämä onnistuu hyödyntämällä ja soveltamalla arkkitehtuurissa animaatio-ohjelmistojen välineitä kuten liike, parametri ja topologia. Tällä tavoin arkkitehtuurin hänen mukaansa lähestyy nestemäisempää muotoa ja siten barokkimaista tyyliä. (Gregg Lynn, 1997)

Vuoden 2010 ”Architectural Design”-lehden numeroa ”Exuberance” voi lukea vuoden 1997 Gregg Lynnin artikkelin valossa. Vuoden 2010 ”Architectural Design”-lehdessä ”Exuberance” esitellään rehevää, kokeilevaa, epärationaalista digitaalisiin menetelmiin ja uuteen teknologiaan nojaavaa tyyliä. Lehdessä suuntausta nimetään sanalla exuberance eli elämäniloksi ja uudeksi virtuositeetiksi. Se on ylenpalttista koristeellisuutta, joka ei perusta järkeen. Lehdessä kirjoitetaan, että suuntaus saa insipraationsa barokista ja rokokoosta. ”Exuberance”-tyyliset kappaleet muistuttavat satunnaisuuteen perustuvia nestemäisiä kappaleita (kuva 3.). (Marjan Colletti, 2010)



Kuva 3.
Nila, Eero Lunden, Markus Wikar, Toni Österlund, 2012
Venetsian biennaaliin 2012 toteutettu komputationaalisia menetelmin toteutettu veistos. Nila-veistos muistuttaa Vuoden 2010 ”Architectural Design”-lehdessä ”Exuberance” mainittua nestemäistä epärationaalista digitaalisiin menetelmiin ja uuteen teknologiaan nojaavaa tyyliä.

Tänä päivänä, vuonna 2018, teknologian avulla voidaan muodostaa ja valmistaa kompleksisia muotoja. Ornamentiikassa mittakaava on muuttunut: esimerkiksi Zaha Hadid arkkitehtuurin edustamista suurista parametrisista linjoista ollaan siirrytty pienempään detaljitason koristeellisuuteen. Ensimmäistä kertaa suunnittelussa ja tuotannossa monimutkaisuus pienessäkään mittakaavassa ei ole samanlainen este, minkälainen se ennen on ollut. Skriptaamalla tai ohjelmoimalla voidaan luoda monimutkaisia kaavoja monimutkaisen geometrian räätälöintiin ja tulostaa suunnitelmat lopulta 3d-printtaamalla. Koristeellisuus on saanut groteskeja muotoja.

Kuva 4.
Verstas Architects, Väre, (kuva otettu) 2017
Kuvan ottohetkellä (03.07.2017) vielä valmisteilla oleva Otaniemen uusi kampusrakennuksen julkisivu on ilmeeltään graafinen.

Michael Hansmeyer on arkkitehti ja ohjelmoija, joka tutkii algoritmien ja tietokoneen käyttöä arkkitehtuurin muodon generoinnissa. Hän on luonnut ”the Sixth Order”-pylvässarjan hyödyntäen 3d-printtausta ja tietokoneteknologiaa (Micheal Hansmeyer, 2013). Pylvässarjassa koostuu monimutkaisista ja yksityiskohtaisista pylväistä, joista jokainen on uniikki kappale.

Helen Castle kirjoittaa ”Architectural Design”-julkaisun ”Patterns of Architecture”-numeron esittelyssä, että graafisuus tai kuviot ovat nousseet nykyarkkitehtuurissa merkittävään asemaan. Esimerkiksi julkisivuissa esiintyy paljon graafisuutta. Julkisivujen graafisuus voi olla kaksiulotteista ornamenttipintaa tai kolmiulotteisia strukturaaleja rakennelmia (kuve 4.). Julkisivujen ohella graafisuus näkyy myös muilla arkkitehtuurin osa-aluille kuten muuan muassa kaupunkisuunnittelussa, maisema-arkkitehtuurissa ja sisustussuunnittelussa (Helen Castle, 2009). Tietokoneet ovat vaikuttaneet oleellisesti dekoratiivisen kuvioiden paluuseen arkkitehtuuriin. Tietokoneilla on mahdollista tehdä näitä aikaisemmin arkkitehtien vieroksuvia toistettavia kuviointeja uudella tavalla, jossa on mahdollisuus toiston lisäksi variaatioon (Mike Silver, 2009).

2.2.3 Suunnittelumallit (Design Patterns) arkkitehtuurissa

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. (Christopher Alexander, 1977)

Arkkitehti Christopher Alexander ja muotoilija John Chris Jones kehittivät Design Pattern-termin (suunnittelumalli) tarkoittamaan jotain valmista ratkaisua usein esiintyvään ongelmaan. Arkkitehtuurissa se tarkoittaa toistuvia käytäntöjä tai toimintamalleja, joiden avulla tietty suunnitteluongelma ratkaistaan. Vuonna 1977 Christopher Alexander julkaisi kirjan ”A Pattern Language”, joka sisältää 253 suunnittelumallia kokonaisista kaupungeista pieniin yksityiskohtiin. Hän yritti luoda suoraan tiettyihin suunnittelutehtäviin sovellettavan suunnittelumetodin. Kirjan kappaleiden nimiä ovat esimerkiksi maaseutukylät, kävelykatu, pohjoisjulkisivu, paksut

seinät ja ruokailuilmapiiri. Jokainen suunnittelumalli sisältää ongelman kuvauksen ja tarjoaa ratkaisun kyseiseen ongelmaan. Christopher Alexanderin mukaan rakentaminen seuraa tiettyjä käytäntöjä tai toimintamalleja (Christopher Alexander, 1977). Sittemmin molemmat termin kehittäjät ovat myöntäneet, että vaikka heidän tarkoituksensa on ollut parantaa suunnittelua, suunnittelumetodeihin pohjautuva liike on lisännyt jäykkyyttä suunnitteluprosessissa ja huonontanut suunnittelulaatua (C.Thomas Mitchell, 1993).

Christopher Alexander ja John Chris Jones lanseeraama käsite Design Pattern on siirtynyt arkkitehtuurista ohjelmistotekniikan yleiseksi käsitteeksi. Suunnittelumalleilla voidaan ratkaista oliopohjaisissa järjestelmissä usein toistuvan ongelman. Suunnittelumalli sisältää aina mallin nimen, ongelman, ratkaisun ja seuraukset (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1997). Suunnittelumalleja voi tehdä ja käsitellä olioiden avulla olio-ohjelmointikielillä.

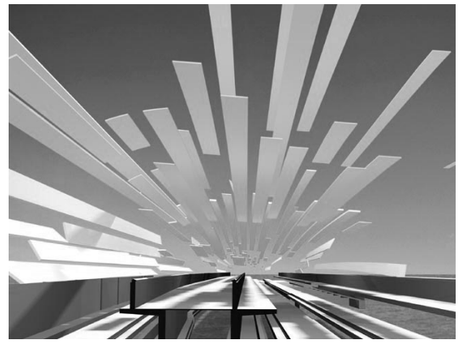
”A Pattern Language”-kirja on arkkitehtuuriteorian klassikko. Kirjalla on löydetty jonkinlainen yhteneväisyys ohjelmistotekniikan ja arkkitehtuurin välille. Teknologian avulla esimerkiksi olio-ohjelmoinnilla voidaan luoda kaavoja ongelman ratkaisemiseksi. Jos arkkitehtuurissa jokainen yksittäinen design-elementti tulisi suunnitella yksitellen, se vaatisi liikaa työtä. Teknologiaa hyödyntäen voimme luoda suunnittelumalleja objektien hallitsemiseksi (Malcolm McCullough, 2008). Arkkitehtuuriin sopii hyvin suunnittelumalleihin perustuva rakenne, sillä kaikkia rakennuksia yhdistää monet yhteiset piirteet. Kaikissa rakennuksesta on eroteltavissa esimerkiksi ulkokuori ja sisätila. Rakennuksissa on useita ikkunoita ja ovia. Toistuvuutta ja toistuvien rakenteiden tilannekohtaisia eroja voidaan ratkaista olio-pohjaisilla skriptaus- ja ohjelmointikielillä.

2.2.4 Generatiivinen ja parametrinen suunnittelu

Generatiivinen suunnittelu

Generatiivinen suunnittelu (kuvat 5. ja 6.) on suunnitteluprosessin tapa optimoida tai etsiä kappaleen muotoa tai muuta ominaisuutta. Se tarkoittaa suunnitelman iterointia joidenkin sääntöjen tai ehtojen puitteissa. Tietokoneella voidaan generoida paras mahdollinen lopputulos tai joukko vaihtoehtoja, joista suunnittelija lopuksi valitsee parhaan mahdollisen suunnitelman. Tietokoneelle asetettavat ehdot voivat esimerkiksi olla visuaalisia, rakenteellisia, taloudellisia tai kontekstikohtaisia ohjeita tietokoneen generaattoriin. Generatiivisessa suunnittelussa tarkalla ehtojen asettamisella on tärkeä rooli.

1970-luvun puolessa välissä Philip Steadman ja Robin Liggett onnistuivat luomaan toimivan ja tehokkaan generaattorin tuottamaan pieniä suorakaiteen muotoisia pohjapiirustuksia. Generaattori optimoi suunnitelman mittasuhteet ja mitat parhaiksi mahdollisiksi alkuehtojen mukaan (William Mitchell, 1990). Myös luonnossa tapahtuu generoitumista. Luonnossa morfogeneesin, eli biologisen rakenteen muodostuessa, organismin muoto ja rakenne kehittyvät; toisin sanoen generoituvat organismin kapasiteetin asettamin rajaehdoin ulkoisten voimien ja -vaikutusten alaisena.



Kuvat 5., 6.
ShinMinamatan Station, Makoto Sei Watanabe, 2004
ShinMinamatan aseman kattosäleet on generoitu optimaaliin varjostuksen, äänen, tuulen ja sateen mukaan.

Parametrinen suunnittelu

Patrik Schumacher on julistanut parametrismin (eng. parametricism) arkkitehtuurin tyylisuuntaukseksi, joka on seurannut postmodernismia ja modernin arkkitehtuurin tyylisuuntauksia. Parametarismi perustuu parametrisiin malleihin, jotka voivat olla mittakaavassa tektonisista yksityiskohdista aina kaupunkisuunnittelun mittakaavaan tai huonekaluista globaaleihin tuotteisiin. Parametriset mallit ovat dynaamisia ja systemaattisia malleja, jotka tukevat muuntelua ja erottelua. (Patrik Schumacher, 2008)

Parametri tarkoittaa kappaleen jotakin ominaisuutta, jota voidaan muuntaa. Tietotekniikassa parametri on arvo, joka syötetään ohjelmalle tai funktiolle. Parametrinen suunnittelu usein määritellään mallintamiseksi kappaleiden suhteiden avulla. Parametri voi esimerkiksi olla viivan pituus, seinän korkeus tai ikkunan koko. Parametriseen suunnitteluun usein yhdistetään deterministinen assosiatiivisuus eli lineaarisuus sen sijaan, että kappaleiden suhteet toimisivat parametriassa rekursiivisesti. Rekursiiviset ominaisuudet määrittyvät niiden itsensä kautta. Parametrisen ja generatiivisen suunnittelun uskotaan lähestyvän toisiaan tulevaisuudessa. (David Jason Gerber, 2014).

Yksinkertaiset tai lineaariset riippuvuussuhteet eri ominaisuuksien välillä eivät tue arkkitehtuurissa systeemiteorian ajatuksia ja niillä on mahdotonta tehdä monimutkaisia evolutiivisia rakenteita. Systeemiteoria on filosofi, biologi Ludwig von Bertalanffy kehittämä teoria siitä, että mikään luonnossa ei elä eristyksessä tai yksinkertaisessa riippuvuussuhteessa: Kokonaisuus on enemmän kuin vain osiensa summa. Arkkitehtuuriin tämä Bertalanffyn luoma kokonaisvaltainen systeemiajattelu on periytynyt alun perin arkkitehtuuriin arkkitehti Christopher Alexanderin kautta 1960-luvulla. Hänen mukaansa suunnittelussa ongelmia tai haasteita ei voi ratkaista yksitellen, vaan suunnittelu on kokonaisvaltainen prosessi. Osia, joista arkkitehtuuri rakentuu, tulee käsitellä holistisesti. (Achim Menges, Sean Alhquist, 2011)

Suunnittelijoilla on tänä päivänä lisääntyneet mahdollisuudet käsitellä erilaista dataa kasvavan teknologian ansioista. Suunnittelijoilla voi olla esimerkiksi mahdollisuus arvioida ja hyödyntää ihmisten käyttäytymistä ja mielipiteitä sekä liikkuvaa elävää dataa. Datan määrän kasvu johtaa monimutkaisempiin suunnitteluehtoihin suunnitteluprosessissa. Suunnittelijalta vaaditaan taitoa rakentaa suunnitteluongelma ja järjestää annettujen ehtojen suhteita. Graafiset ohjelmointiympäristöt, ohjelmien pluginneina eli liitännät ja niiden kirjastot sekä skriptaus- tai ohjelmointikielet ovat tehneet mahdolliseksi yhdistää generatiivinen tekijä parametriin, jolloin suunnitelmaa voidaan arvioida ja visualisoida nopeasti. (David Jason Gerber, 2014).

2.2.5 Muotoutuminen

Sana topologia on johdettu kreikasta ja tarkoittaa ”paikan tietoa” tai ”paikan tuntemusta”. Topologia on matematiikan osa-alue, joka tutkii muotoja ja kuvioita. Topologiassa ollaan kiinnostuneita muotojen ja kuvioiden kvalitatiivista ominaisuuksista ja etenkin sellaisista ominaisuuksista, jotka säilyvät jatkuvissa muutoksissa. Topologia käsittelee pistejoukkoja, joita kutsutaan topologisiksi avaruuksiksi. Perinteinen anekdootti on, että ”Topologi on matematiikka, joka ei erota kahvikuppia munkkirinkilästä”. Topologiassa neliö, suorakaide, ympyrä ja ellipsi ovat topologisesti ekvivalentteja. Ympyrä, jonka keskellä on reikä, ei kuitenkaan ole ekvivalentti edellisten kanssa. (Arttu Ojanperä, 2013)

Vuonna 1995 John ja Julia Frazerin julkaiseman ”An Evolutionary Architecture”-kirjan ydinajatus on, että arkkitehtuuri on elävä ja kehittyvä asia. Arkkitehtuuri ja rakennukset eivät ole vain staattisia kokonaisuuksia, jotka ovat irrallaan kontekstistaan, vaan nekin kehittyvät ja muuttuvat ympäristön vaikutuksista (kuva 7.). Tämä on ollut sinänsä ilmiselvä ajatus: Kulttuuri, aika ja käyttäjät ovat aina muokanneet rakennuksia. John Frazer kuitenkin kirjassaan ja 1970-luvulla alkaneissa tutkimuksissaan pyrkii oikeasti määrittelemään arkkitehtuuria, jolla on eläviä ja kehittyviä ominaisuuksia. ”An Evolutionary Architecture”-kirjan mukaan, vaikka arkkitehtuuri on keinotekoista, sille voidaan yrittää luoda luonnon ominaisuuksia. Arkkitehdin on tarkoitus rakentaa ympäristöä, joka on symbioottisessa suhteessa ja metabolisessa tasapainossa kontekstinsa kanssa. (Gordon Pask, 1995)

John Frazer on yksi edelläkävijä tuodessaan tietokoneteknologiaa arkkitehtuuriin (Manuel Kretzer, 2017). Tietokone ja ohjelmointi tai skriptaus olivat John Frazerille tapa syventyä evolutiivisiin prosesseihin arkkitehtuurissa. Se ei sinänsä ollut itsetarkoitus, mutta Frazerille se oli välttämättömyys, jotta luonnon prosessien jäljittely onnistuisi. Monet luonnon rakenteet muistuttavat joillain tasolla muodostuessaan algoritmisia prosesseja. Molemmat seurailevat vähintään samankaltaisia matemaattisia kaavoja. Algoritmeilla voidaan generoida ominaisuuksia tai osia joidenkin sääntöjen puitteissa. Niillä voi luoda toistoa ja muutoksia. Frazer hyödynsi tietokonetta ja koodi-skriptausta muodostaakseen biologisia prosesseja kuten morfogeneesi, evoluutio, geneettinen koodi, replikaatio ja valinta. (John Frazer, 1995)

Luonnosta voidaan löytää hämmästyttävän paljon matemaattisia prosesseja tai algoritmeja seuraavia rakenteita. Evolutiivisessa suunnittelussa algoritmeilla on keskeinen rooli (Mark Burry, 2010). John Frazer algoritmien avulla stimuloi prototyyppejään. Hänen tarkoituksena oli generoida arkkitehtuurin muotoa ja massaa ja luoda virtuaalisia arkkitehtuurimalleja. Kirjassa ”An Evolutionary Architecture” etsitään yhtäläisyyksiä ja eroja arkkitehtuurin - ja luonnon prosessien väliltä. (John Frazer, 1995)

”An Evolutionary Architecture”-kirja on vaikuttanut arkkitehtuuriin ja arkkitehtuurin teoriaan. Arkkitehtuuri on elävä ja mukautuva organismi, joka vastaanottaa tietoa ympäristöstään ja reagoi siihen. Yhtäläisyyksiä evolutiivisen arkkitehtuurin kanssa voidaan löytää esimerkiksi kineettisestä arkkitehtuurista, ympäristöön reagoivien materiaali- ja itsekasaantuvien rakennusosien tutkimuksesta.

Kuva 7.
Öljysäiliö 468, (kuva otettu) 2017
Helsingin Kruununvuorenrannan valotaideteoksen reagoin ympäristön olosuhteisiin. Valojen rytmiä säätelee tuulennopeus. Alkuillasta valot palavat valkoisena ja illan myöden valot muuttuvat punaiseksi.

Kineettinen arkkitehtuuri tarkoittaa arkkitehtuuria, jossa rakenteelliset osat tai osa niistä ovat liikuteltavia kineettisesti. Kineettisessä arkkitehtuurissa rakenteet voivat esimerkiksi reagoida johonkin ympäristössä tapahtuvaan muutokseen ja sopeutua siihen, joilloin kineettiset osat toimivat adaptiivisesti. Adaptiivinen tarkoittaa mukautuvaa ja kineettinen liikkuvaa. Esimerkkejä kineettisestä julkisivuratkaisustia arkkitehtuurissa ovat vuonna 2012 Etelä-Korean maailmannäyttelyyn valmistunut Soma-arkkitehtien suunnittelema paviljonki (kuva 8.) ja Abu Dhabin vuonna 2012 valmistuneet Al Bahr-tornitalot. Soma-arkkitehtien paviljongissa on hengittävä julkisivu, jonka lamellit kiertyvät elektronisesti säädelläkseen sisätilan ilmaa. Sen sijaan Abu Dhabin tornitalojen julkisivurakenteen kolmionmuotoiset varjostimet sulkeutuvat suojatakseen rakennusta voimakkaimmalta auringonpaisteelta.

Kuva 8.
Soma, One Ocean, 2012
One Oceanin julkisivu hengittää. Lamellit ovat suljettavissa yhtenäiseksi julkisivupinnaksi.

Abu Dhabin tornitalot ja Soma-arkkitehtien suunnittelema paviljonki mukautuvat kineettisesti ympäristön olosuhteiden vaikutuksesta. Kineettisesti toimivia rakenteita voidaan säädellä adaptiivisesti kybernetiikan avulla. Kybernetiikka tarkoittaa erilaisten järjestelmien säätöjä ja kommunikaatiota tutkivaa tiedettä. Tiede tutkii itseohjautuvia automaattisesti ohjautuvia järjestelmiä. Tiedettä voidaan soveltaa biologisiin, yhteiskunnallisiin ja laitteiden järjestelmien tutkintaan. Abu Dhabin tornitaloissa ja Soma-arkkitehtien suunnittelemassa paviljongissa sekä mukautuminen että kybernetiikka voidaan johtaa ”An Evolutionary Architecture”-kirjan teoriaan.

Frazerin ”An Evolutionary Architecture”-kirjassa on kybernetiikkaa lähellä olevia näkökulmia (Jules Moloney, 2011) Esimerkiksi ”An Evolutionary Architecture”-kirjassa mainitaan palautemekanismia kuvaava termi ”iteratiivinen adaptaatio”, joka voidaan johtaa kybernetiikan alle kuuluvaksi käsitteeksi (Jules Moloney, 2011). Gordon Pask on ollut tärkeä vaikuttaja kybernetiikan tuomisessa arkkitehtuuriin (Achim Menges, Sean Alhquist, 2011). Hän ohjasi John Frazeria ”An Evolutionary Architecture”-kirjan kanssa. Kirja alkusanat ovat Gordon Paskin kirjoittamat.

2.2.6 Ohjelmien avoin jakaminen ja suunnitelmien omistajuus

Ohjelmia tai skriptejä hyödyntävä tapa luoda arkkitehtuuria asettaa kysymyksen suunnittelijan oikeudesta suunnitelmaan, joka on luotu jotakin ohjelmaa, skriptiä tai generaattoria käyttämällä. Ohjelmia hyödyntämällä arkkitehdin rooli aktiivisena suunnittelija hämärtyy. Missä

menee raja, että arkkitehdin sijasta ohjelma ratkaisee suunnitteluongelman? Kostas Terzidisin mukaan perinteisesti arkkitehtuurin keskustelussa ja tuotannossa dominoiva paradigma, eli arkkitehtuurissa yleisesti hyväksyttä ajattelutapa, on ollut ihmisen intuitio ja nerokkuus. Algoritmiavusteinen suunnittelu on hyödyntänyt metodeita, joille ei ole vastaavia edeltäjiä. Terzidisin mukaan, jos arkkitehtuuri suuntaa algoritmiseen suuntaan, suunnittelumetodeihin tulisi sisällyttää komputationaalisia menetelmiä, milloin tietokone voisi generoida muotoja ja suunnitelmia, jotka ovat ymmärryksemme ulkopuolella. (Kostas Terzidis, 2003)

Toisaalta ohjelmat saattavat myös rajoittaa suunnitteluratkaisuja, jos suunnittelija on liian sidottu ohjelmaan eikä tunne sen prosesseja. Suunnittelija tekee suunnitteluratkaisuja ohjelman tarjoamilla välineillä, vaikka mahdollisia välineitä voisi olla muitakin. CAD-ohjelmistojen konventionaalinen käyttötapa saattaa rajoittaa suunnittelijan luovuutta. Ohjelmistoihin sidottu suunnittelija käyttää ainoastaan ohjelman toimintoja, joita ohjelmisto tarjoaa eikä ymmärrä muita tietokoneen mahdollisuuksia. Cooper Unionin professori Pablo Lorenzo-Eiroa toteaa ”Form on the Relationship between Digital signifiers and Formal Autonomy”-esseessään, että jos arkkitehdit eivät murra tai korvaa ohjelmien tai käyttäjäliittymien lähdekoodeja luodakseen omiaan, on selvää, että heidän työtään määrittää kulloisenkin alustan ominaisuudet. Ohjelmistot, alustat ja koodit määrittävät arkkitehdin taiteelliselle työlle kehyksen, jonka sisällä arkkitehti operoi.

Generaattoreita käyttämällä arkkitehdin rooli aktiivisena suunnittelija on vähintään teoriassa mahdollista hävittää. Toisaalta generaattoreilla on mahdollista luoda ennalta-arvaamatonta geometriaa ja tuloksia ja paljastaa mahdollisia suunnitteluvaihtoehtoja, jotka eivät tulisi muuten ilmi. Suunnittelijan rooli generaattoreita hyödyntäessä on suunnitteluehtojen asettamisessa. Arkkitehti Adam Doulgerakist toteaa Athina Theodoropouloun diplomityössä, että on vaikea tuottaa tarkoituksenmukaista lopputulosta, ellei suunnittelija ole ohjelmiston (tai generaattorin) luoja itse tai erinomaisesti tunne ohjelmiston (tai generaattorin) rakenteita ja toimintoja.

Sekä generaattoreihin, skriptien ja CAD-ohjelmistoihin hyödyntämiseen liittyy kysymys tekijänoikeuksista. Generointityökaluja hyödyntäen arkkitehdin on teoriassa mahdollista suoraan generoida valmis suunnitelma. Esimerkiksi 1970-luvulla Philip Steadman ja Robin Liggett generoivat pohjaratkaisuja suoraan hyödyntäen generaattoria (William Mitchell, 1990). Myös Lorenzo-Eiroan kirjoittaa, että jos alusta, esimerkiksi CAD-ohjelmisto, määrittää kehyksen työskentelylle, suunnittelijan voidaan ajatella vain tulkitsevan alustan vaihtoehtoja, joita alusta ehdottaa. Hänen mukaansa tämä voi johtaa siihen, että tietyssä tilanteessa ohjelmoija voisi väittää omistavansa ohjelmalla luoman geometrian tekijänoikeudet (Pablo Lorenzo-Eiro, 2013). Se, kuka omistaa generoimalla tai ohjelmistoa käyttämällä tehdyn suunnitelman voidaan yrittää määrittää suunnittelijan käyttämän työkalun tarkoituksenmukaisuudella. Arkkitehti Anna Laskarin toteaa Athina Theodoropouloun diplomityössä generaattoreista, että se, kuinka hyvin tunnet prosessia generaattorin taustalla lähteestä tulokseen, määrittää roolisi aktiivisena suunnittelija lopputuloksessa. (Athina Theodoropoulou, 2007)

Yleisesti ottaen skriptauskulttuuriin kuuluu avoimuus ja skriptien jako. Kuitenkin moni skriptaaja kopioi skriptejä toisiltaan nykyajan avoimuuden nimiin niin, että alkuperäiset skriptin kirjoittajat eivät saa tunnustusta skripteistään. Turha varovaisuus skriptien kanssa on kuitenkin usein ylimitoitettua. Todennäköisyys, että joku on jo kirjoittanut kirjoittamasi skriptin jossain

päin maailmaa ja laittanut sen jakoon on suuri. Mark Burry kirjoittaa ”Scripting Cultures ”-julkaisussa, skriptaajat tai ohjelmoijat voidaan jakaa kahteen sukupolveen, joista toinen jakaa avoimesti tietoaan mukaan lukien koodinsa ja toinen, joka pelkää pakkomielteisesti muiden varastavan heidän ideansa. Hugh Whitehead Foster+Partner arkkitehdeiltä toteaa ”Scripting Cultures”-kirjassa (Mark Burry, 2010):

At present scripters tend to be of the”lone gun” mentality and are justifiably proud of their firepower usually developed many late nights of obsessive concentration. There is the danger that if celebration of skills is allowed to obscure and divert from the real design objectives, then scripting degenerates to become an isolated craft than developing an integrated art form.

Avoimuus on johtanut siihen, että skriptien levitessä osa niistä kulutetaan loppuun niin, että niiden innovatiivisuus menettää vaikuttavuutensa. Ramsgard Thomsen toteaa Burryn haastattelemana, että moni oppii skirptaamaan muiden jakamien koodien kautta. Tässä on vaarana joidenkin skriptien toistuvuus. Moni skriptaaja vain levittää toisten koodeja sen sijaan, että kirjoittaisivat omia ja harva skriptaaja on todella innovatiivinen. John Frazer sanoo ”Scripting Cultures”-kirjassa, että innovatiivisessa skriptauksessa tai ohjelmoinnissa ei olla kauheasti päästy pidemmälle sitten 1960-1970-luvulta. (Mark Burry, 2010)

Modernin arkkitehtuurin historiassa rakennuksen taiteellinen työ on tunnistettu arkkitehdin persoonan kautta (Athina Theodoropoulou, 2007). Välineet eivät saisi vaikuttaa työskentelyyn, vaan ennemmin luoda vaihtoehtoja ja mahdollisuuksia suunnittelijalle. On tärkeä ymmärtää, mitä voi tehdä eikä kuinka. Skriptaus ja ohjelmointi auttavat ymmärtämään, arvioimaan ja kriittisesti tietokoneen tai ohjelmiston prosesseja. Suunnitteluprosessissa arkkitehdin tulee osata valita paras mahdollinen työkalu. Parhaimmillaan kuitenkin ohjelmilla ja generaattoreilla on kaikki potentiaali kasvattaa suunnittelijan luovuutta tarjoamalla vaihtoehtoja. Välineet tulee vain tuntea.

2.2.7. Skriptauksen ja Grasshopperin toiminnalliset erot

Graafisella skriptieditorilla tarkoitetaan graafista ohjelmointiympäristöä, jossa skriptiä kootaan tekstimuodon sijaan graafisesti. Arkkitehtien hyödyntämät Grasshopper ja Dynamo ovat esimerkiksi graafisia skriptieditoreja. Graafiset skriptieditorit perustuvat niin sanottuihin flowchart-diagrammeihin eli vuokaavioihin, jotka esittävät algoritmia tai prosessia komponenttien ja komponentteja yhdistävien nuolien tai johtimien avulla. Vuokaavioiden avulla myös selitetään ja analysoidaan ohjelmien prosesseja.

Jo vuonna 1986 Fraderick P. Brooks kirjoitetussa tunnetussa esseessään ”No-Silver-Bullet - Essence and Accident in Software Engineering” kritisoi graafisia skriptieditoreja todeten, että ne tuskin tulevat ikinä olemaan vakuuttavia ohjelmointiympäristöjä. Brooksin mukaan graafisten skriptieditorien vuokaaviot vastaavat huonosti ohjelmoinnin rakenteita: Sovelluksia tai ohjelmia on vaikea visualisoida. Sovellukset rakentuvat yhden kaavion sijaan yleensä useisiin eri kaavioihin. Brooksin mukaan vuokaaviot tuntuvat liian yksiulotteisilta esittämään moniulotteisesti verkottunutta skriptiä. Lisäksi Brooksin mukaan tietokoneiden ruudut ovat yksinkertaisesti

liian pieniä pikseleiltään, jotta pystytään tarkastelemaan detaljoitua vuokaavio-diagrammia. Tietokoneen ruudulta ei pysty näkemään kuin kapean siivun diagrammista kerrallaan, ja kokonaisuudesta on siten jo vaikea saada käsitystä. (Frederick P. Brooks, 1986)

Tukholman teknillisen yliopiston KTH:n tutkija Pablo Miranda Carranzan mukaan Fraderick P. Brooksin kritiikki on edelleen validia graafisia skriprieditoreita, kuten esimerkiksi Grasshopperia, kohtaan. Pablo Miranda Carranzan mukaan Grasshopperin toiminta on kaukana niin sanotun täydellisen Turingin-koneen periaatteista (Turing-täydellinen). Turingin-koneen avulla selitetään usein ohjelmoitavan tietokoneen toimintaa. Se on teoreettinen malli algoritmien havainnollistamiseksi. Turingin-kone on äärellinen automaatti, jossa on äärettömän pitkä lukunauha, lukunauhaa lukeva lukupää ja äärellinen määrä tiloja. Automaatti pystyy liikkumaan oikealle ja vasemmalle, pysähtymään, lukemaan nauhaa ja korvaamaan nauhalta luvun ja tulostamaan sen tilalle toisen. Koneelle syötetään ohjeita ja syötetyn ohjeen luenta päättyy, kun kone saavuttaa Halt-tilan. Turing-täydellinen pystyy laskemaan minkä tahansa koneelle syötetyn tehtävän. Tämä tarkoittaa sitä, että tietokonekieli voi ilmaista minkä tahansa loogisen väittämän. Turingin-koneen avulla voidaan esimerkiksi esittää luonnonilmiöiden kausaliteetteja. (Pablo Miranda Carranza, 2017)

Turingin-kone ja korkean tason ohjelmointikielet kuten esimerkiksi Python toimivat niin sanotulla control-flow-periaatteella. Sen sijaan Grasshopper toiminta perustuu data-flow-periaatteeseen, jossa dataa siirretään lähde-komponentista kohde-komponenttiin (Pablo Miranda Carranza, 2017). Grasshopperissa rakennetaan vuokaaviota, jossa ei ole mahdollisuutta takaisinkytkentään kohde-komponentista takaisin lähteeseen. Yhteen Grasshopper tiedostoon on mahdotonta luoda verkottunutta skriptiä. Control-flow:ssa ohjelma pystyy toisella tavalla kuin data-flow:ssa esimerkiksi ongelman ratkaisuun, tekemään päätöksiä, luomaan branching(haarautuvia)-, -loop(silmukoituvia)- ja rekursiolauseita. Control flow:n luomat verkot voivat olla moniulotteisempia kuin mitä esimerkiksi Grasshopperin graafinen komponenttiliitäntä vasemmalta oikealle on kykenevä rakentamaan.

Jokaista tietokoneohjelmaa oli sitten kyse Rhinosta, Grasshopperista, Revitistä, tai jostain aivan muusta, on jossain vaiheessa rakennettu tekstimuotoisella ohjelmoinnilla tai skriptauksella. Korkean tason ohjelmointikielet ovat ilmaisuvoimallisempia kuin mitä Grasshopper tai muut graafiset skriptieditorit ovat. Myös Grasshopperiin usein, jos esimerkiksi ohjelmaan kaivataan tiettyä funktionaalisuutta, liitetään plugineita eli liitäntöjä, jotka joku on luonut tekstimuotoisesti ohjelmoimalla tai skriptaamalla. (Pablo Miranda Carranza, 2017) Grasshopper on saavuttanut kuitenkin suosion arkkitehtien parissa, mikä kertoo ohjelman tehokkuudesta.

3. TUTKIELMAT

3.1 SKRIPTAUKSET

Tässä osassa esittelen kolme skriptaustutkielmaa. Jokainen tutkielma sisältää selostuksen, 3d-printatun mallin, skriptin ja rendauksia. Skripti eli koodi sisältää myös skriptin kommentit, joita merkitään skriptiin risuaitamerkinnällä. Kommenteissa selitän koodin sisältöä ja prosessia: Kuinka skripti on muotoutunut.

Jokaisella tutkielmalla on oma suunnittelukohteensa. Lisäksi käsittelen joka tutkielmassa jotakin skriptaukseen liittyvää piirrettä. Ensimmäisessä tutkielmassa käsittelen satunnaisuutta eli random-toimintoja. Toisen tutkielman aiheenani on attraktori eli esimerkiksi kahden objektin välimatkan vaikutus objektien ominaisuuksiin. Kolmannessa tutkielmassa tutkin pisteiden paikan määräytymistä xyz-avaruudessa. Kaikissa skripteissä on rekursiivisia toimintoja. Skriptejä voi hyödyntää työkaluna esimerkiksi karniisin-, akustiikkapaneelin- ja näyttelytilan kattorakenteen suunnittelussa.

Käytän ohjelmointiympäristönä seuraavissa harjoituksissani Grasshopperin komponenttiin sidottua Python-editoria. Seuraavia harjoituksien lopputulosta ei siis saa ulos esimerkiksi kääntämällä koodi suoraan Rhinon ohjelmointiympäristöön RhinoPythonEditoriin, vaan vaatii oikeat muuntujat Grasshopperin komponentin syötteeseen ja tulokseen ja kenties erillisen koodin purkamaan sisäkkäisiä tietorakenteita, joita Grasshopperin yksittäinen komponentti ei pysty käsittelemään. Grasshopperin komponentit eivät ymmärrä sisäkkäisiä tietorakenteita eli nested- listoja. Nested-listat joudutaan purkamaan Grasshopperissa.

Tämän osan tavoitteena on luoda työkaluja, jota Rhinon, Revitin tai Archicadin oletustyökalut eivät tarjoa. Työkalujen tarkoitus on tutkia skriptaamalla algoritmein saavutettavaa geometriaa. Skriptit ovat syntyneet useiden kokeilujen tuloksena.

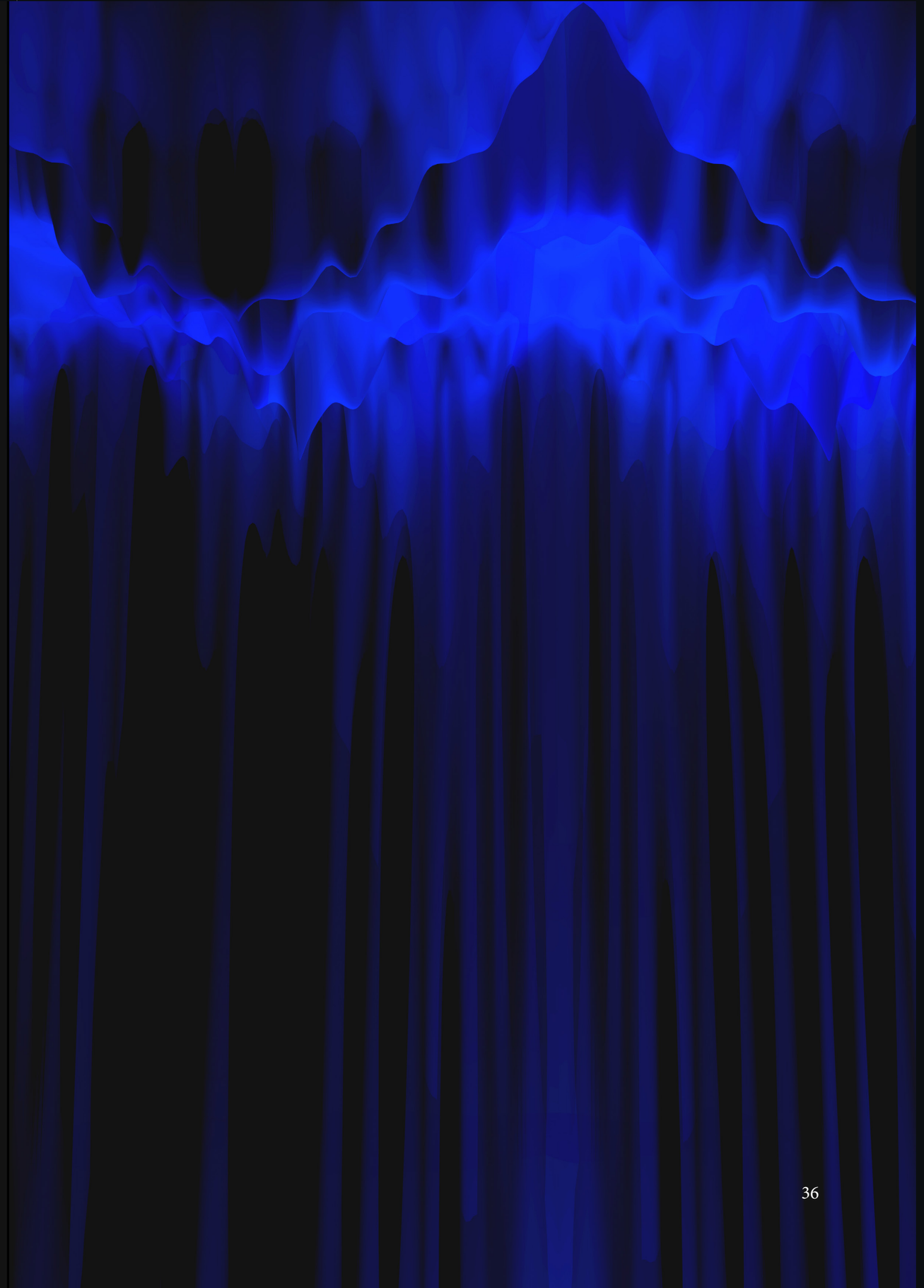
3.1. Karniisin suunnittelu

Sana random viittaa yleensä ennalta-arvaamattomaan tapahtumaan. Ensimmäisessä tutkielmassa olen hyödyntänyt random-generaattoria. Generaattori ei oikeasti perustu satunnaislukuihin, vaikka siltä saattaa näyttääkin. Satunnaistoiminnot perustuvat näennäissatunnaislukuihin. Nämä pseudosatunnaisluvut seuraavat tiettyjä tunnettuja algoritmeja, jotka saavat toiminnon näyttämään satunnaisilta. Näennäissatunnaisluvut generoituvat seed-luvusta. Kaikki satunnaisluvut, jotka pohjautuvat seed-lukuun ovat lopulta ennustettavia.

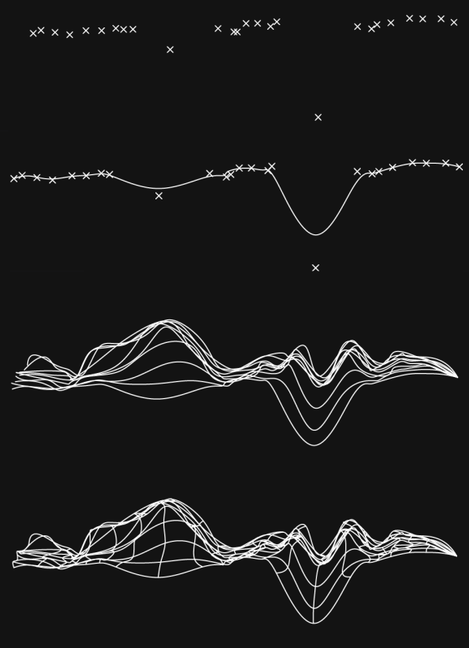
Tietokoneen luomilla satunnaisluvuilla saadaan aikaan näyttäviä efektejä ja pystytään rikkomaan näennäisesti tietokoneen kaavamaista jäykkyyttä. Satunaisuusgeneraattori on tullut tunnetuksi generatiivisen taiteen piirissä, missä tietokone ohjelmoidaan generoimaan satunnaisotannalla algoritmeista taidetta. Satunnaistoimintoihin tulee suhtautua hiukan kriittisesti: Lähemmällä katsannolla satunnaistoiminnot näyttävät kuitenkin vain umpimähkäisiltä.

Tässä harjoituksessa olen piirtänyt y-akselin suuntaista sin-käyrää, jonka kertoimet vaihtelevat satunnaisesti. Kerroin kymmenosan todennäköisyydellä ajoittain monikertaistuu piirtäen syvän uuman käyrään. Ensimmäistä sin-käyrää monistetaan x-akselin suuntaisesti algoritmisesti siten, että joka monistuskerralla syvät uumat kaventuvat ja kohoavat reunoistaan. Lopputuloksena on orgaaninen nestemäinen groteski satunnaisgeometria.

Arkkitehti voi hyödyntää satunnaisgenerointia karniisin suunnittelussa. Satunnaisgeneroinnilla saadaan aikaan näyttävää orgaanista geometriaa arkkitehtuurin koristeaiheisiin.



3.1. Karniisin skriptin kommentit

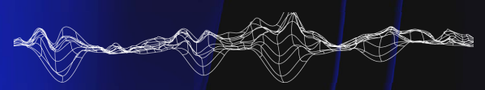


Kuva 9.
1. tutkielman vaiheita
Tutkielmassa geometria muodostuu käyriä yhdistävästä pinnasta. Ensin harjoituksessa luodaan koordinaatistoissa samalle x-arvolle sijoittuva pistelista. Pistelista z-arvot seurailevat sinikäyrää. Sinikäyrän kerroin vaihtelee jokaisen uuden luodun pisteen kohdalla. Samoin pisteiden asettuminen y-akselin suunnassa vaihtelee: Pisteet eivät jakaudu y-akselin suunnassa tasaisesti. Skriptin toisessa funktiossa pistelistaa lähdetään kopiomaan uudella x-arvolla muuntamalla joka kerta myös pisteiden yz-arvoja. Lopuksi yhdistetään pisteet käyräksi ja käyrät yhdistetään pinnoiksi.

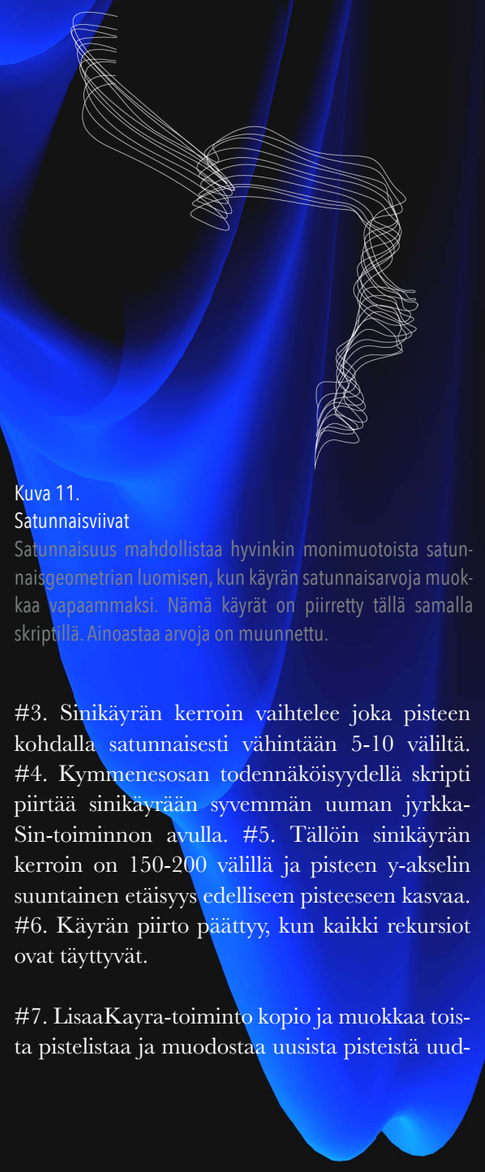
#1. PiirraSinKayra-toiminto luo pisteet geometrian uloimmasta käyrää varten. Pisteet lisätään pistelistaan a. #2. A-listan pisteiden x-arvot ovat aina nolla. Y-arvot ovat piin kertoimia ja z-arvot ovat sinikäyrän arvoja muuttujalla y. Pisteiden y-arvot ovat aina sinikäyrän huippuja.

3.1. Karniisin skripti

```
import rhinoscriptsyntax as rs
from Rhino.Geometry import Point3d
import random
import math

def piirraSinKayra (firstPoint, rekursio): #1.

    siirraY = random.uniform(5, 10)* math.pi #2.
    siirraZ = (math.sin(siirraY))*random.uni
    form(-10, 10)
    siirraX = random.uniform(0, 0) #3.

    def jyrkkaSin (firstPoint, rekursio, rekursio2, \
    siirraZ, siirraY):
        if rekursio2 <= 2:
            point = firstPoint + Point3d(siirraX, \
            siir raY/2, siirraZ)
            uusisiirraY = (random.uniform(40, 50)*\
            math.pi)
            uusisiirraZ = -abs((math.sin(siirraY))*\
            random.uniform(150, 200)) #2.
            a.append(point)
            jyrkkaSin (point, rekursio+1, rekursio2+\
            1, uusisiirraZ, uusisiirraY)
        elif rekursio2 <= 3:
            uusisiirraY = random.uniform(5, 10)* \
            math.pi
            uusisiirraZ = -abs((math.sin\
           (uusisiirraY))*random.uniform(-10, 10))
            firstPoint[2]=math.sin(uusisiirraY)*\
            random.uniform(-10, 10)
            point = firstPoint + Point3d(siirraX, \
            uusisiirraY, uusisiirraZ)
            a.append(point)
            return piirraSinKayra (firstPoint, \
            rekursio+1)

    point = firstPoint + Point3d(siirraX, siirraY, \
    siirraZ)
    if rekursio < 1:
        a.append(point)
        piirraSinKayra (point, rekursio+1)
    elif rekursio < 20: #4.
        vaihtoehdot = range(0, 10)
        valinta = random.choice(vaihtoehdot) #5.
        if not valinta == 1:
            a.append(point)
            piirraSinKayra (point, rekursio+1)
        elif valinta == 1:
            jyrkkaSin (firstPoint, rekursio, 0, siirraZ,\
            siirraY)


firstPoint = rs.coerce3dpoint(startPoint)
a = [] #pistelista
b = []
c = []
d = []
e = []
g = []
h = []
```

Kuva 10.
1. tutkielman vaiheita
Lisäten PiirraSinKayran-toiminnon rekursiokertoja geometrian pituutta voidaan kasvattaa loputtomasti. Vaikka pituutta kasvatetaan, skripti luo samaan algoritmiin perustuvaa satunnaisgeometriaa.

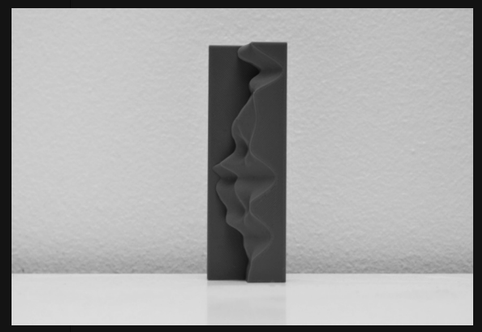


Kuva 11.
Satunnaisviivat
Satunnaisuus mahdollistaa hyvinkin monimuotoista satunnaisgeometrian luomisen, kun käyrän satunnaisarvoja muokkaa vapaammaksi. Nämä käyrät on piirretty tällä samalla skriptillä. Ainoastaa arvoja on muunnettu.

#3. Sinikäyrän kerroin vaihtelee joka pisteen kohdalla satunnaisesti vähintään 5-10 väliltä. #4. Kymmenesosan todennäköisyydellä skripti piirtää sinikäyrään syvemmän uuman jyrkka-Sin-toiminnon avulla. #5. Tällöin sinikäyrän kerroin on 150-200 välillä ja pisteen y-akselin suuntainen etäisyys edelliseen pisteeseen kasvaa. #6. Käyrän piirto päättyy, kun kaikki rekursiot ovat täyttyvät.

#7. LisaaKayra-toiminto kopio ja muokkaa toista pistelistaa ja muodostaa uusista pisteistä uud-

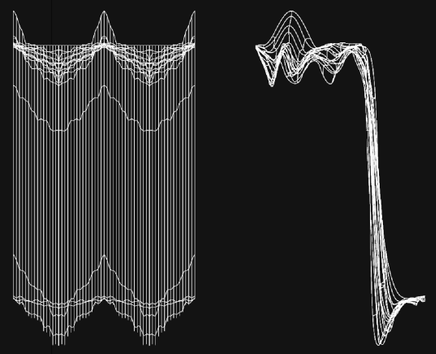
```
i = []
j = []
k = []
l = []

piirraSinKayra (firstPoint, 0)

def lisaaKayra (jasenA, a, c): #6.
    siirraY = random.uniform(-2, 2)
    siirraZ = random.uniform(-2, 2)
    if jasenA < len(a):
        if jasenA == 0:
            uusiPiste = a[jasenA] + Point3d(-20, \
            siirraY, siirraZ)
            c.append(uusiPiste)
            lisaaKayra (jasenA+1, a, c)
        elif jasenA < len(a)-1:
            uusiPiste = a[jasenA] + Point3d(-20, \
            siirraY, siirraZ)
            etaisyys1 = a[jasenA-1][1]-a[jasenA-2][1]
            etaisyys2 = a[jasenA][1] - a[jasenA-1][1]
            if etaisyys2 < math.pi*10:
                if a[jasenA+1][1] - a[jasenA][1] > \
                math.pi*10:
                    uusiPiste = a[jasenA] + Point3d(-20, \
                    siirraY+20, siirraZ)
                    c.append(uusiPiste)
                    lisaaKayra (jasenA+1, a, c)
                else:
                    c.append(uusiPiste)
                    lisaaKayra (jasenA+1, a, c)
            elif etaisyys2 > math.pi*10: #7.
                if etaisyys2 > math.pi*10 and etaisyys1 > \
                math.pi*10:
                    uusiPiste = a[jasenA] + Point3d(-20,\
                    siirraY-20, siirraZ)
                    c.append(uusiPiste)
                    lisaaKayra (jasenA+1, a, c)
                elif etaisyys1 < math.pi*10:
                    if a[jasenA][2] < -15:
                        uusiPiste = a[jasenA] + Point3d(-20,\
                        0, siirraZ+20)
                        c.append(uusiPiste)
                        lisaaKayra (jasenA+1, a, c)
                    else:
                        uusiPiste = a[jasenA] + Point3d(-20,\
                        siirraY, siirraZ)
                        c.append(uusiPiste)
                        lisaaKayra (jasenA+1, a, c)
    elif jasenA < len(a):
        uusiPiste = a[jasenA] + Point3d(-20, \
        siirraY, siirraZ)
        c.append(uusiPiste)
        lisaaKayra (jasenA+1, a, c)


lisaaKayra(0, a, c)
lisaaKayra(0, c, d)
lisaaKayra(0, d, e)
lisaaKayra(0, e, g)
lisaaKayra(0, g, h)
lisaaKayra(0, h, i)
lisaaKayra(0, i, j)
lisaaKayra(0, j, k)
```



Kuva 12.
3d-printatut malli 1. harjoituksesta
Kuvassa yksi skriptin generoinneista. Pinnat käyrien välille on lisätty Grasshopperissa loft-työkalulla.

en pistelistan uudella x-arvolla. #8. Toiminnon avulla syviä sinikäyrän uomia kohotetaan ja kavennetaan sen reunoja. Jos kahden pisteen y-akselin suuntainen etäisyys on suurempi kuin 10 x pii, skripti siirtää ensimmäistä pistettä lähemmäs toista pistettä ja kohottaa pisteitä 20 yksikköä z-akselin suuntaisesti ylöspäin. Täten syvät sinikäyrän muodostamat uomat alkavat vähitellen kuroutua umpeen.

Satunnaistoiminnoilla skripti tuottaa aina erilaisen geometrian skriptiä käännettäessä.



Kuva 13.
1. tutkielman vaiheita
Rendattavan mallin lähtöpiste on siirretty siten, että skripti luo mallista verhomaisen seinämän .


```
lisaaKayra(0, k, l)

viivat = rs.AddCurve(a)
viivat2 = rs.AddCurve(c)
viivat3 = rs.AddCurve(d)
viivat4 = rs.AddCurve(e)
viivat5 = rs.AddCurve(g)
viivat6 = rs.AddCurve(h)
viivat7 = rs.AddCurve(i)
viivat8 = rs.AddCurve(j)
viivat9 = rs.AddCurve(k)
viivat10 = rs.AddCurve(l)

b.append(viivat)
b.append(viivat2)
b.append(viivat3)
b.append(viivat4)
b.append(viivat5)
b.append(viivat6)
b.append(viivat7)
b.append(viivat8)
b.append(viivat9)
b.append(viivat10)
```

3.2. Akustiikkapaneelin suunnittelu

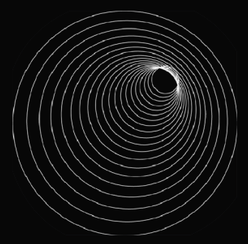
Attraktori tarkoittaa ohjelmoituja pisteitä, jotka hylkivät tai vetävät puoleensa toisia pisteitä tai objekteja. Attraktorin keskeinen ominaisuus on sen etäisyys toisesta pisteestä. Mitä lähempänä toinen piste attraktoria on, sitä enemmän attraktori vaikuttaa toiseen pisteeseen. Attraktorin ei tarvitse vaikuttaa nimenomaisesti sen ja toisen objektin väliseen välimatkaan vaan attraktori voi vaikuttaa esimerkiksi sen läheisyydessä olevien objektien kokoon tai väriin.

Attraktori luodaan laskemalla etäisyys attraktorin ja objektin välillä. Etäisyyksistä voidaan luoda lista, joka järjestellään uudelleen suurimman tai pienimmän arvojen mukaisesti. Objektin jokin ominaisuus, oli se sitten koko tai väri, voidaan kalibroida uudelleen järjestellyn etäisyyslistan mukaisesti. Näin syntyy attraktori-efekti.

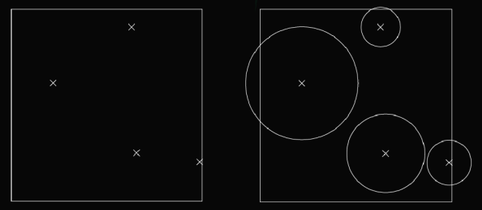
Tutkielmassa renkaiden satunnaiskeskipisteet ovat aina vähintään viidenkymmenen yksikön etäisyyksillä toisistaan. Jos lähimmän renkaat ulkokehä on liian lähellä toisen renkaan keskipistettä, piirtyy keskipisteeseen rengas, joka ei kosketa viereistä rengasta. Satunnaisrenkaiden erilaisia joukkoja voidaan generoida kääntämällä skripti uudelleen.

Arkkitehti voi hyödyntää attraktoria akustiikkapaneelien suunnittelussa. Hän voi määrittää satunnaisesti paneelien paikkaa sekä määrittää paneelien maksimi- ja minimietäisyyksiä toisistaan attraktorien avulla sekä generoida satunnaisotannalla erilaisia paneelivaihtoehtoja.

3.2. Akustiikkapaneelin skriptin kommentit



#1. Koodin alkuun lisätään kaikki tarvittavat kirjastot ja viittaukset muihin tiedostoihin import-käskyllä. Tämä helpottaa huomattavasti koodarin työmäärää, sillä jokaista funktiota ei tarvitse luoda itse. Skriptiin ollaan lisätty rhinoscriptsyntax - ja random - kirjastot sekä poimittu Rhino.Geometry-kirjastosta yksi toiminto.



Kuva 14.
2. tutkielman randomYmpyrat - funktion vaiheet 1 ja 2.
Ensin funktio arpoo 200x200 yksikön alueen sisälle yksitellen pisteiden paikat siten, että ne ovat vähintään 60 yksikön etäisyydellä toisistaan. Vaiheessa 2 funktio luo listan keskipisteiden ympärille muodostuvien kehien säteistä. Sinänsä randomYmpyrat ei vielä piirrä ulommaisia kehiä keskipisteisiin.

#2. Määritetään skriptiin toiminto, joka #3. luo satunnaispisteen 200x200x10 yksikön alueelle. Toimintoa voidaan kutsua uudelleen ja jokainen kutsukerta lisää pisteen listaan a ja piirtää uuden pisteen määritellylle alueelle. Se, kuinka monta kertaa kutsua luetaan, säädellään rekursioparametrin avulla. #7. Tässä skriptissä rekursiokerrat on määritetty alle viiteen.

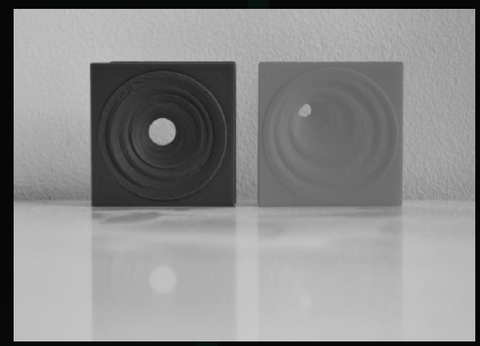
#4. Ensimmäiselle rekursiokerralla pisteelle ei asetu piirtoaluetta lukuun ottamatta vaatimuksia. Sitä seuraavina kutsukertoina uusien pisteiden tulee kuitenkin olla vähintään 60 yksikön

3.2.Akustiikkapaneelin skripti

```
import rhinoscriptsyntax as rs
from Rhino.Geometry import Point3d
import random #1.

def randomYmpyrat (firstPoint, rekursio): #2.

    siirraX = random.uniform(0, 200)
    siirraY = random.uniform(0, 200)
    siirraZ = random.uniform(0, 10)   #3.

    point = firstPoint + Point3d(siirraX, siirraY, \
    siirraZ)

    def pisteet (point):

        def etaisyysLaskenta(point, x):
            if x < len(a):
                etaisyysX = rs.Distance(point, a[x])
                etaisyysLista.append(etaisyysX)
                valiArvo = etaisyysX - koot[x]
                kokoArvio.append(valiArvo)
                etaisyysLaskenta (point, x+1)
        etaisyysLista = []
        kokoArvio = []

        etaisyysLaskenta(point, 0)

        if  all(arvot > 60 for arvot in etaisyysLista): #4.
            a.append(point)
            etaisyydet.append(etaisyysLista)
            if min(kokoArvio) < 80:
                kokoX = random.uniform(20, \
                min(kokoArvio))
            else:
                kokoX = random.uniform(20, 80)
            koot.append(kokoX)
            randomYmpyrat(firstPoint, rekursio +1)

    if rekursio < 1:
        a.append(point)
        etaisyys1 = rs.Distance(point, a[0])
        etaisyydet.append(etaisyys1)
        koko1 = random.uniform(30, 60)
        koot.append(koko1)
        randomYmpyrat(firstPoint, rekursio + 1 )
    elif rekursio < 2:
        etaisyys2 = rs.Distance(point, a[0])
        if etaisyys2 > 60:
            a.append(point)
            etaisyydet.append(etaisyys2)
            if etaisyys2 - koot[0] < 80: #5.
                koko2 = random.uniform(20, (etaisyys2-\
                koot[0]))
            else:
                koko2 = random.uniform(20, 60) #6.
            koot.append(koko2)
            randomYmpyrat(firstPoint, rekursio + 1)
    elif rekursio < 5:  #7.
        pisteet (point)
```


etäisyyksillä muista jo piirretyistä pisteistä. Jos tämä ei toteudu, piste jää piirtymättä. #13 Etäisyyksien laskentaa varten kokoan skriptiin etäisyydet-listaa RandomYmpyrat-funktion ulkopuolelle.

#6 Funktiossa randomYmpyrat määritetään myös pisteen ympärille muodostuvan uloimman kehän koot. Uloimman kehän säde on normaalisti jotain 20 ja 80 yksikön välillä. #5. Jos jonkin rekursion keskipiste kohdistuu alle 80 yksikön etäisyydelle lähimmästä toisen määrittyneen

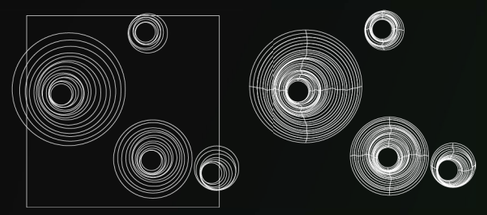


Kuva 15.
3d-printatut mallit kahdesta rengasrykelmästä.
Skriptillä saan aikaiseksi orgaanista pintaa. Näiden kahden mallin skriptissä on pieni ero: vasemman puoleisessa mallissa sisäkkäisten kehien keskipiste siirtyy jokaista kerralla kehää piirtäessä satunnaisesti. Oikean puoleisessa ensimmäistä kehää seuraavien sisempien kehien keskipiste määrittyy satunnaisesti, mutta säilyy, joka piirto kerralla samana.

pisteen kehästä, uusimman kehän säteen mahdollinen maksimi on sen keskipisteen etäisyys lähimmästä kehästä. Siten kehät voidaan määrittää skriptissä siten, että ne eivät koskaan leikkaa toisiansa.

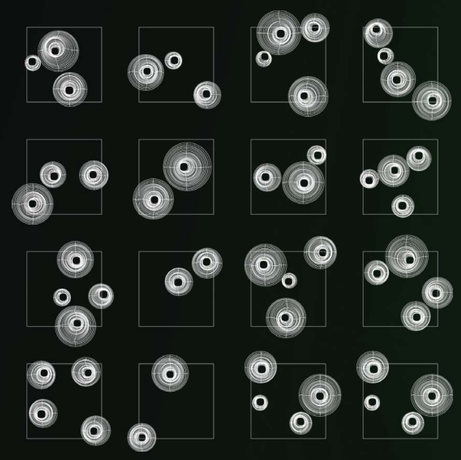
#8. Muodostetaan ympyrärykelmä-luokka, jonka parametreja ovat satunnaiskeskipisteet ja niiden uloimman kehän koot. Luokan avulla voidaan luoda keskipisteisiin ns. olioita. #14.

```
class Ympyrarykelma: #8.

    def __init__(self, piste, lista):
        self.piste = piste
        self.lista = lista

    def ympyroidenPiirto(self, kokolista): #9.
        jasenA = self.piste
        koko = kokolista
        def uusiYmpyra(jasenA, koko):
            if koko >= 10:
                siirraX = random.uniform(-2, 2) #10.
                siirraY = random.uniform(-2, 2)

                liikutaX = random.uniform(190, 200)
                liikutaY = random.uniform(0, 0)
                liikutaZ = random.uniform(-5, 5)

                liikuta1X = random.uniform(0, 0)
                liikuta1Y = random.uniform(190, 200)
                liikuta1Z = random.uniform(-5, 5)

                piste1 = [liikutaX, liikutaY, liikutaZ]
                piste2 = [liikuta1X, liikuta1Y, liikuta1Z]

                plane = rs.PlaneFromFrame(jasenA,\
                piste1,
                piste2 ) #11.
                ympyra2 = rs.AddCircle( plane, koko )
                uusiPiste = jasenA + Point3d(siirraX, \
                siirraY, -1)
                uusiKoko = koko * \
                random.uniform(0.89, 0.9)
                self.lista.append(ympyra2)
                uusiYmpyra (uusiPiste, uusiKoko)

        def pintojenLuonti(ympyraJasen): #12.
            if ympyraJasen < len(self.lista) -1:
                ympyroidenYhdistaminen =[self.lista[/
                ympyraJasen], self.lista[ympyraJasen+1]]
                pinta = rs.AddLoftSrf(/
                ympyroidenYhdistaminen)
                pinnat.append(pinta)
                pintojenLuonti(ympyraJasen+1 )

        uusiYmpyra(jasenA, koko)
        pintojenLuonti(0)

firstPoint = rs.coerce3dpoint(startPoint)

a = []
b = []
c = []
d = []
e = []
f = []

pinnat = []
etaisyydet = [] #13.
koot= []

randomYmpyrat(firstPoint, 0)

if len(a) <= 1:
```



Kuva 16.
2. tutkielman luokan metodit ympyroidenPiirto ja pintojenLuonti
YmpyroidenPiirto-metodi piirtää kehät lähtötietonaan randomYmpyrat-toiminnon parametrit, keskipiste sekä uloimman kehän säde. Kehien kaltevuus, keskipiste ja koko vaihtelevat. Rekursio päättyy, kun kehän koko saavuttaa kymmenen yksikköä. PintojenLuonti-metodi luo kehien väliin pinnat.

Kutsun näitä olioita skriptin lopussa piste1, piste2, piste3 jne. #9. Luokat sisältävät luokan sisäisiä toimintoja eli metodeita. YmpyröidenPiirto - metodi YmpyräRykelmä luokassa muodostaa uloimman kehän sisäpuolelle uusia pienempiä kehiä. Kehien muodostus loppuu, kun kehän koko saavuttaa alle kymmenen yksikön.



Kuva 17.
2. tutkielman generointeja
Skriptin avulla voidaan generoida lukemattomasti eri vaihtoehtoja skriptin asettamien ehtojen puitteissa.

```
    piste1 = Ympyrarykelma(a[0], b) #14.
    piste1.ympyroidenPiirto(koot[0])
elif len(a) <= 2 :
    piste1 = Ympyrarykelma(a[0], b)
    piste2 = Ympyrarykelma(a[1], c)
    piste1.ympyroidenPiirto(koot[0])
    piste2.ympyroidenPiirto(koot[1])
elif len(a) <= 3:
    piste1 = Ympyrarykelma(a[0], b)
    piste2 = Ympyrarykelma(a[1], c)
    piste3 = Ympyrarykelma(a[2], d)
    piste1.ympyroidenPiirto(koot[0])
    piste2.ympyroidenPiirto(koot[1])
    piste3.ympyroidenPiirto(koot[2])
elif len(a) <= 4:
    piste1 = Ympyrarykelma(a[0], b)
    piste2 = Ympyrarykelma(a[1], c)
    piste3 = Ympyrarykelma(a[2], d)
    piste4 = Ympyrarykelma(a[3], e)
    piste1.ympyroidenPiirto(koot[0])
    piste2.ympyroidenPiirto(koot[1])
    piste3.ympyroidenPiirto(koot[2])
    piste4.ympyroidenPiirto(koot[3])
elif len(a) <= 5:
    piste1 = Ympyrarykelma(a[0], b)
    piste2 = Ympyrarykelma(a[1], c)
    piste3 = Ympyrarykelma(a[2], d)
    piste4 = Ympyrarykelma(a[3], e)
    piste5 = Ympyrarykelma(a[4], f)
    piste1.ympyroidenPiirto(koot[0])
    piste2.ympyroidenPiirto(koot[1])
    piste3.ympyroidenPiirto(koot[2])
    piste4.ympyroidenPiirto(koot[3])
    piste5.ympyroidenPiirto(koot[4])
```

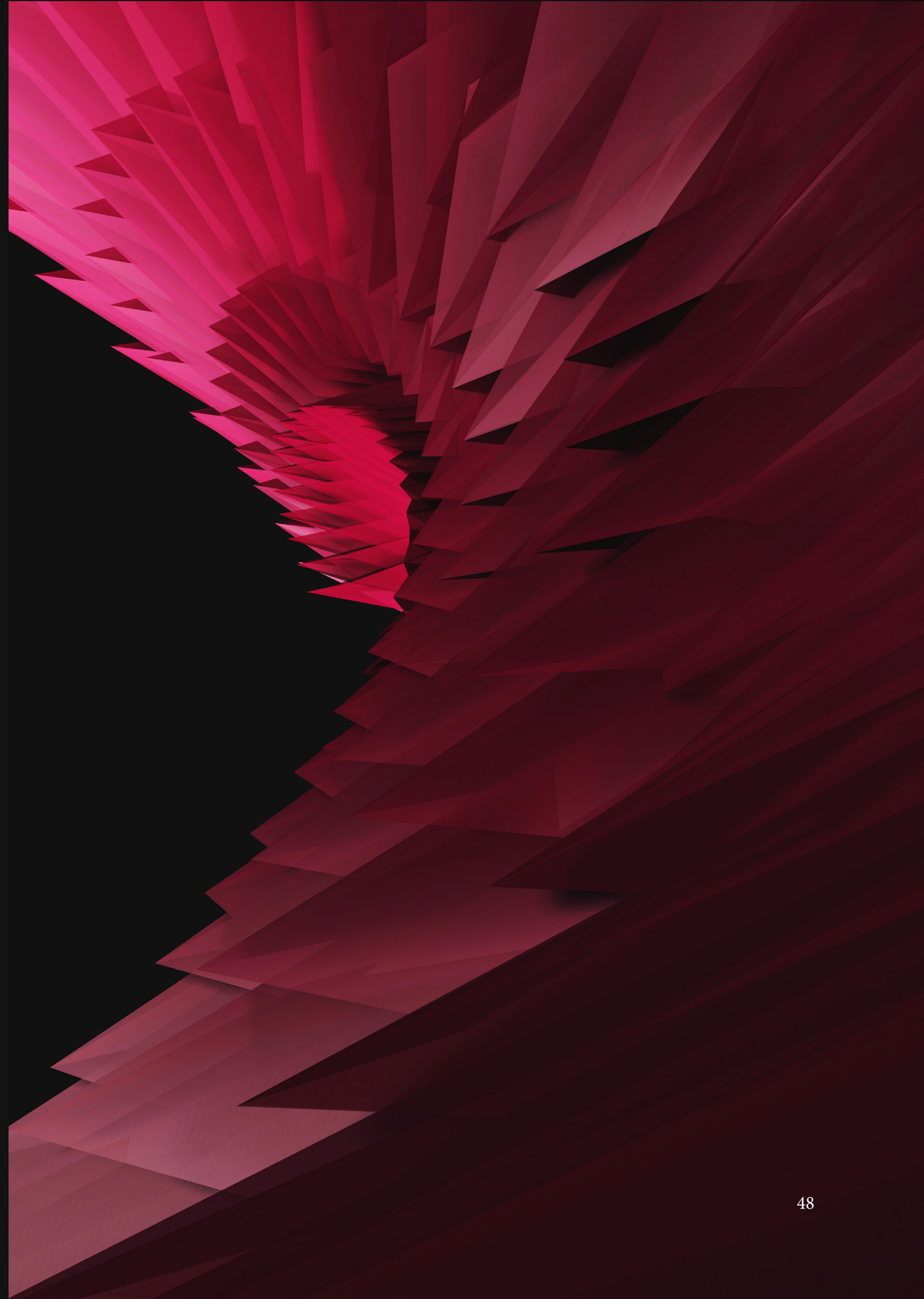
#10. Uloimman kehän sisäpuolelle muodostuvien uusien renkaiden keskipiste siirtyy hiukan jokaiselle piirtokerralla satunnaisesti x- ja y-suunnassa ja keskipiste laskee xyz-koordinaatistossa yhden yksikön alaspäin z-akselin suuntaisesti. #11. Myös kehien kaltevuus vaihtelee joka kehän kohdalla. Tämä saa aikaiseksi vaihtelevaa orgaanista pintaa.

3.3 Näyttelytilan kattorakenteen suunnittelu

Kolmas tutkielma muodostuu pisteistä, jotka on koottu spiraalimaisesti kiertyviksi kaariksi. Joka toista kaaren pistettä kohotetaan z-suunnassa ja siirretään y- ja x-suunnassa algoritmisesti, jolloin pisteet yz-tasossa muodostavat sahalaitakuviota. Yhdistäessä pisteet pinnoiksi geometria muistuttaa piikikästä höyhenpeitettä. Skriptissä hyödynnetään trigonometriaa kaaren muodostuksessa. Skriptauksen avulla arkkitehti voi työskennellä hyödyntäen matemaattisia operaattoreita.

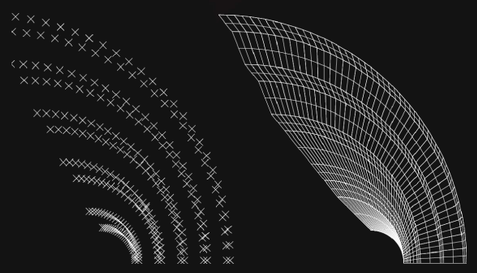
Tietotekniikassa yleensä selitetään hierarkkista tietorakennejärjestelmää, jolla säilötään informaatiota, puumetaforan kautta. Tästä on syntynyt datatree-käsite, jota hyödynnetään myös skriptauksessa. Tietorakenne on juurakko, josta haarautuu oksia, jotka niin ikään säilövät tietoa. Oksistosta voi haarautua lukematon määrä aina vain uusia oksistoja. Oksistojen yhteinen tekijä on niiden juurakko. Eri oksistoilla ei kuitenkaan ole mitään tekemistä toistensa kanssa, ellei ne ole samassa haarautumassa. Datatree käytännössä tarkoittaa skriptatessa nesting-ominaisuutta. Tässä skriptissä hyödynnetään nesting- eli sisäkkäisiä listoja pisteiden paikan hallinnassa ja pisteiden järjestelyssä.

Datatree-rakenteet auttavat arkkitehtiä monimutkaisen geometrian hallinnassa. Seuraavien sivujen skriptiä voidaan hyödyntää esimerkiksi näyttelytilan kattorakenteen suunnittelussa. Hyödyntäen nesting-listoja voidaan kattorakennesuunnittelua edelleen jatkaa tarkempaan detaljisuunnitteluun skriptin avulla. Jokaiseen uniikkiin piikkiin voidaan esimerkiksi lisätä kuhunkin piikkiin sopiva ikkuna. Ikkunan koko voi määrittyä auringon suunnan mukaan generaattorin avulla.



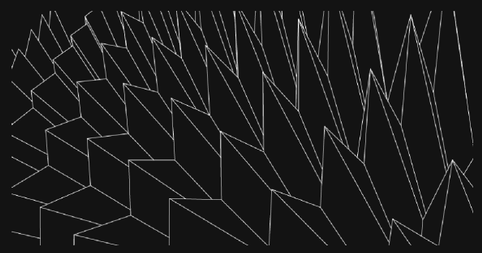
3.3 Kattorakenteen skriptin kommentit

#1. Kirjastojen tuonti koodiin. Math-kirjasto lisää matematiikan operaattoreita koodiin kuten esimerkiksi neliöjuuri ja sini.



Kuva 18.
3. tutkielman yläprojektiot
Skriptin ensimmäinen toiminto luo pisteet kaarelle ja toinen toiminto yhdistää pisteet neljän pisteen ryppäissä pinnoiksi. Ensimmäinen toiminto muodostaa xy-koordinaatistossa neljäsosan r-säteellisestä ympyrän kehästä. Jokaisella kaarella on kaksikymmentäviisi pistettä ja koska, jos y on sin(25x 0.02xpii) = sin(pii/2) = 1 niin kaari päättyy pisteeseen, jossa y saa maksimi arvonsa annetuilla parametreilla. Joka toiselle kaarelle on annettu hiukan eri parametrit ja siitä johtuen joka toinen kaari hipoo edellistä y-arvon lähestyessä nollaa.

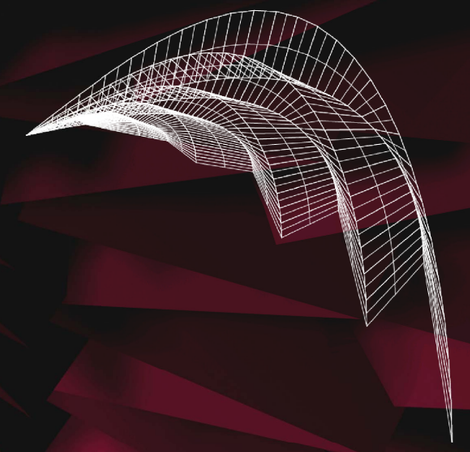
#2. uudetPisteet-toiminto luo kaareutuvaa pistejonoa. Pistejonot muodostavat omat b-listansa. #5. B-lista lisätään toiminnon päättyessään yhtenä nested-listana a-listaan. #6. Täten a-listasta muodostuu datatree, joka on muotoa [[...], [...], [...], ... [...]]. Jokainen nested-lista sisältää 26 xyz-koordinaatistoon sijoitettavaa pistettä.



Kuva 19.
3. tutkielman muunnelma
Kaarien joka toista pistettä on kohotettu z-suunnassa ja siirretty sivusuuntaisesti xy-suunnassa.

3.3 Kattorakenteen skripti

```
import rhinoscriptsyntax as rs
from Rhino.Geometry import Point3d
import random
import math #1.

def uudetPisteet (r, s, kerroin, zKerroin, toinenkerroin, kolmaskerroin): #2.

    b=[]

    for i in range(0,26):

        uusiY = math.sin(kerroin*i)*r #3.
        uusiX = math.sqrt( r**2 -  /
        (uusiY**2))+s #3.
        uusiZ = uusiY*zKerroin*uusiX

        if i % 2 == 0: #4.
            uusiX = math.sqrt(r**2 -  /
            (uusiY**2)) +1.2*kolmaskerroin +2
            uusiY = math.sin(kerroin*i)*r +/
            1.2*toinenKerroin
            uusiZ = uusiY*zKerroin*uusiX /
            + 4*kolmaskerroin*math.sin(i*3.08)/
            piste = Point3d(uusiX, uusiY, uusiZ)
        else:
            piste = Point3d(uusiX, uusiY, uusiZ)

        b.append(piste) #5.

    a.append(b) #6.

a = []
b = []
c = []

pinnat = []


uudetPisteet (20, 8, 0.02*math.pi, 0.005, /
10, 1) #7.
uudetPisteet (30, -0, 0.02*math.pi, 0.01, 9 , 2)
uudetPisteet (50, -8, 0.02*math.pi, 0.005, 8, 3)
uudetPisteet (60,-16, 0.02*math.pi, 0.01, 7, 4)
uudetPisteet (80,-24, 0.02*math.pi, 0.005,/
 6, 5)
uudetPisteet (90,-32, 0.02*math.pi, 0.01, 5, 6)
uudetPisteet (110,-40, 0.02*math.pi, 0.005,/
4, 7)
uudetPisteet (120,-48, 0.02*math.pi, 0.01, /
3, 8)
uudetPisteet (140,-56, 0.02*math.pi, 0.005, /
2, 9)
uudetPisteet (150,-64, 0.02*math.pi, 0.01, /
1, 10)


def pintojenLuonti(jasenA, jasenB): #8.

    if jasenB < 25:
        kaarienYhdistaminen = [a[jasenA]
[jasenB], a[jasenA][jasenB+1], a[jasenA+1]
[jasenB+1], a[jasenA+1][jasenB]] #9.

        pinta = rs.AddSrfPt/
        (kaarienYhdistaminen)
        pinnat.append(pinta)

        if jasenA < 8:
            pintojenLuonti(jasenA+1, jasenB)

        else:
            pintojenLuonti(0, jasenB+1)

pintojenLuonti(0,0) #10.
```



Kuva 20.
3. tutkielman sivuprojektio
Z-arvot määrittyvät kertomalla skriptissä x- ja y-arvot yhteen. Lopputuloksena saadaan aikaiseksi spiraalimaisesti kaartuvia pintoja.

Kuva 21.
3d-printatut malli 3. harjoituksesta
3d-printattu malli muistuttaa joitakin luonnossa esiintyvää rakennetta. Luonto noudattelee hämmästyttävän hyvin matematiikan lainalaisuuksia. 3d-printauksesta on jätetty skriptiin kirjoitetut piikit pois.

#3. Lisätään pisteitä xy-koordinaatistosta kaarelle, jonka säde on r ja keskipiste koordinaatiston origossa. Kaaret ovat neljäsosa ympyrän kehästä. Lisätään pisteet kaarelle trigonometriaa hyödyntäen. #7. uudetPisteet-toimintoa toistetaan eri parametreilla säätäen esimerkiksi sädettä ja xyz-arvoja.

#4. Viimeisessä tulostusversiossa kappaleen pintaan muodostuu piikkejä. Joka toista kaaren pistettä siirretään z-suunnassa ylöspäin, sekä maltillisemmin xy-suunnassa, jotta piikit eivät osoittaisi suoraan ylöspäin. Korkeutta säätelee niin ikään sini ja kerroin.

#8. PintojenLuonti-toiminto yhdistää a-listan pisteet pinnoiksi #9. valiten aina kaksi vierekkäistä pistettä yhdeltä kaarelta ja kaksi pistettä ensimmäistä kaarta seuraavalta toiselta kaarelta. #10. PintojenLuonti-funktio kutsuu toimintoa.

4. LOPPUPÄÄTELMÄT

Pablo Mirandaa Carranzaa lainaten on tärkeää pohtia, kuinka ohjelmointi tai skriptaus vaikuttavat arkkitehtuuriin (Pablo Mirandaa Carranza, 2017). Skriptaus arkkitehtuurin metodina on kiinnostava ja tärkeä aihe. Ohjelmointia hallitseva sukupolvi on kasvamassa ja hakeutuu jossain vaiheessa arkkitehtiopintojen pariin. Arkkitehtuuriin kasvaa sukupolvi, joka hallitsee ja ymmärtää ohjelmointia ja skriptausta.

CAD-ohjelmistoissa on jotain perustavanlaatuisesti muututtava. Tarvitaan entistä monimuotoisempi työkaluja, jotka yhdistävät suunnittelua ja tuotantoa. Samalla koodilla, jolla suunnittelet, tulisi voida myös automatisoida tuotantoa useassa eri järjestelmässä. Tietomalleista tulee entistä monimutkaisempia, kun malliin voidaan yhdistää tietoa tuotantomenetelmistä. Toisaalta arkkitehtuurin alalla on myös kysyntää ohjelmille ja ohjelmistoille, jotka joustavat paremmin kuin nykyiset ohjelmistot suunnittelijan tarpeiden mukaan. Kaivataan ohjelmistoja, joita voi yhdistää, ohjelmistojen välisiä työkaluja tai ehkä yksi yhteinen, vahva, helposti opittava skriptauskieli. Ohjelmistojen tulee tukea paremmin omien työkalujen luomista tai tuomista ohjelmiin.

Tässä diplomityössäni olen tutkinut skriptausta arkkitehdin metodina. Olen syventynyt metodin ominaisuuksiin ja taustaan arkkitehdin näkökulmasta. Diplomityössä keskeisiä käsitteitä ovat olleet parametri, generointi, rekursio, oliot, yhtälöt, evoluutio, topologia ja silmukointioperaattorit. Näin lopuksi, on tärkeää erotella, puhutaanko skriptaus-metodista arkkitehdin työskentely- vai arkkitehtuurin muotoilun tutkimustapana, jotta voimme pohtia skriptauksen etuja arkkitehtuurin muotoilussa.

Skriptaus arkkitehdin työskentelytapana

Skriptaus soveltuu arkkitehdille metodiksi arkkitehtuurin muotoiluun, ja lisää mahdollisuuksia verrattuna joihinkin CAD-ohjelmistoihin. Skriptaus on kuitenkin ennen kaikkea tapa luoda arkkitehdille työkaluja ja tehostaa hänen työskentelyään. Skriptaus tulee yleensä arkkitehdille kyseeseen vasta silloin, kun mitään muuta keinoa ongelman ratkaisuun ei ole. Edut verrattuna ohjelmistoihin ovat skriptauksella tällä hetkellä ja näillä työkaluilla vähäiset. Nykyiset arkkitehdeille suunnatut ohjelmistot sopivat hyvin staattisten kappaleiden suunnitteluun.

Skriptaus on tehoton työskentelymuoto kappaleiden muotoiluun. Ohjelmistoihin valmiiksi rakennetut työkalut ovat arkkitehtejä varten, jotta arkkitehtien ei tarvitsisi työskennellä skriptaamalla. Geometriat, joihin käytetyimpien CAD-ohjelmistojen työkalut eivät sovellu, on parasta pyrkiä ratkaisemaan graafisilla skriptieditoreilla ennen tekstimuotoista skriptausta.

Sktiptauksella voidaan kuitenkin automatisoida ja tehostaa arkkitehdintyötä. Arkkitehtitoimistoissa on tarvetta koodaajille. Suomessa tiedän, että esimerkiksi arkkitehtitoimisto B&M:llä on töissä koodaaja.

Mark Burry kirjoittaa, että erityisesti isoissa toimistoissa koodarit, jotka skriptaavat parantaakseen toimiston rutiineja ja tehokkuutta, ovat arvostettuja työntekijöitä. Toimiston tehokkuuden parantaminen skriptauksen avulla katsotaan arvokkaammaksi työksi kuin skriptaaminen efektien luomiseksi. Praktiikassa arkkitehdin saattaa kuitenkin olla vaikea hyödyntää tai vastaanottaa toimiston koodarin taitoa toimiston kiireisessä arjessa, vaikka lopulta skriptaus saattaisi joissakin tilanteissa parantaa toimiston toimintaa. (Mark Burry, 2011)

Skriptaus tutkimustapana arkkitehtuurin muotoilussa

Skriptauksen tai ohjelmoinnin suuri etu on sen riippumattomuus ohjelmistoista. Ohjelmointi on luovin ja vapain tapa työskennellä tietokoneitse. Se auttaa ajattelemaan ja suhtautumaan kriittisesti ohjelmistoihin.

Skriptaus muotoilumetodina on kiinnostavillaan ja parhaimmillaan silloin, kun sillä tutkitaan liikettä, muutosta, muotoutumista ja prosesseja. Skriptauksen avulla voidaan esimerkiksi rakentaa monimutkaisia ja monia vaikuttavia tekijöitä huomioon ottavia generaattoreita.

4.1 LOPUKSI

Tämä diplomityö on tarkoittanut minulle vapautumista CAD-ohjelmistoista. Tietokone on vain väline, eikä ohjelmisto saa määritellä, sillä luotua muotoilua tai arkkitehtuuria. Sinänsä tässä ajatuksessa ei ole mitään uutta. Ensimmäisenä arkkitehtuurin opintovuotenani meitä nuoria arkkitehtiopiskelijoita kehotettiin työskentelemään käsin piirtämällä ja fyysisin mallein. Meitä ei haluttu ahtaa tietokoneen ohjelmistojen luomaan muottiin eikä vaivata ohjelmistoihin liittyvillä haasteilla. Opinnoissa aina ohjattiin, että tietokone on vain apuväline, ja että kyllä keinot keksitään, miten monimutkainenkin suunnitelma saadaan tietokoneitse esitettyä.

En ole varma, menivätkö kuitenkaan kaikki opit läpi, ja uskoinko. Mielestäni, jossain vaiheessa tietokone kavensi työskentelytapojani ja ideointiani. Jouduin miettimään, kuinka saan työni tietokoneella toteutettua. Joitakin algoritmisia ideoitani en pystynyt toteuttamaan enkä kokeilemaan, koska minulle ei yksinkertaisesti ollut työskentelytapoja niihin. Nyt koen olevani vapautunut näistä haasteista. Uskon osaavani ja tiedän, kuinka paljon tietokoneitse on tehtävissä. Diplomityö on antanut minulle itsevarmuutta työskennellä tietokoneella ja tämän jälkeen, olen entistä vakuuttuneempi, että kyllä keinot keksitään.

Minulla on ollut vaikea määritellä, onko tämä diplomityö suunnittelu- vai tutkimuspainotteinen työ. Heti alkuun minulle oli selvää, että aihe, skriptaus arkkitehtuurin metodina tunnetaan kansainvälisellä tasolla hyvin, ja minun on vaikeaa lähteä täysin nollasta tuomaan aiheeseen jotain uutta kiinnostavaa näkökulmaa. Siksi lähdin tekemään suunnittelutyötä. Aluksi valokuvasin luontoa, yritin mallintaa luonnon algoritmeja ja suunnittelin erilaisten generaattorien laatimista. Työ meni moneen kertaan uusiksi matkan varrella. Työ säilyi kuitenkin suunnittelutyönä aivan loppumetreille saakka. Aivan viime metreillä päädyin kuitenkin nimeämään työn tutkimuspainotteiseksi.

Diplomityö on ollut minun tutkimusmatkani arkkitehtiopiskelijana ohjelmoinnin maailmaan. Se alkoi pohdinnasta, saavuttaako skriptaamalla arkkitehtuurin muotoiluun jotain uutta. Opettelin matkan varrella itse skriptaamaan ja tutustuin skriptauksen operaattoreihin. Diplomityöni on ollut yleissivistävä projekti. Opin ymmärtämään matkan varrella, mihin CAD-ohjelmistot perustuvat ja skriptaus on opettanut minulle, että tietokoneella työskentely on parhaimmillaan hauskaa ja luovaa työtä. Diplomityössäni olen saanut kehittää suurimpia mielenkiintoni kohteitani arkkitehtuurissa, jotka ovat muotoilu, massoittelu ja pienoismallit.

Tämä diplomityöni toimii myös minun omana mielipiteenäni, mihin suuntaan arkkitehtuuriopintoja tulee mielestäni kehittää. Teknologian ymmärtäminen on tulevaisuudessa keskeinen kansalaistaito. Haluan kannustaa tällä diplomityöllä myös muita arkkitehtiopiskelijoita ja arkkitehtejä ohjelmoinnin pariin.

LIITTEET

TUTKIELMIEN METODI

Tässä liitteessä esittelen diplomityön tutkielmaosan skriptien rakentamiseen tarvittavia työkaluja. Käyn läpi lyhyen katsauksen Python-ohjelmointikielen perusteisiin ja tapaan, jolla rakennan tutkielmat.

Ohjelmointi tarkoittaa tietokoneelle tai jollekin muulle laitteelle annettavia toimintaohjeita. Tietokone käsittelee vain bittijonoja, jota kutsutaan konekieleksi, ja tällä kielellä toimintaohjeita syötetään tietokoneelle. Koska konekieli on hankalaa ja virhealtista, lähes kaikki ohjelmointi tehdään yleensä lausekielellä, joka on ohjelmoijalle helpompi kieli ymmärtää kuin bittijonot. Lausekielellä ohjelmoija luo lähdekoodia eli skriptiä. Lähdekoodi muodostuu komennoista, joka muunnetaan kääntäjällä tai tulkilla tietokoneelle bittijonoksi. Lausekieltä sellaisenaan tietokone ei ymmärrä. Yleensä ohjelmointiympäristö tarjoaa kääntäjän. Ohjelmointiympäristö on alusta, jossa ohjelmoija tai skriptaaja luo ohjelmia. Yksinkertaisimmin se on tekstieditori.

Esittelen alkuun ohjelmointi- tai skriptikieliä. Päädyin valitsemaan tutkielmieni kieleksi Pythonin sen yleisyyden vuoksi. Se, että mitä kieltä käytän tässä diplomityössä ei ole merkittävää. Ohjelmointi tai skriptausta harrastavien viesti on yleensä se, että skriptikielen valinta ei ole tärkeää, vaan se, että vain aloittaa jostakin. Muiden kielten oppiminen on helpompaa sitten, kun hallitsee jo yhden. Kielet muistuttavat toisiaan rakenteiltaan ja toiminnoiltaan. Pythonin on laajalle levinnyt, monikäyttöinen, joustava ohjelmointikieli, josta aloittelijakin oppii alkeet varsin nopeasti. Python on yksi käytetyimmistä ohjelmointikielistä.

Skriptauksen harjoitteluun tarvitaan ohjelmointiympäristö tai tekstieditori sekä kääntäjä. Aloitan harjoitteluun pienistä harjoituksista, siirryn isompiin ja lopulta toteutan tutkielmat. Pythonin yksinkertaisten ohjelmien kirjoittamiseen voi aloittaa esimerkiksi IDLE-python editorilla. Internetistä löytyy myös paljon ladattavia erilaisia ohjelmointiympäristöjä.

Ohjelmointikieliä:

Python

Python on suosituimpia ohjelmointikieliä. Sen on kehittänyt Guido van Rossum vuonna 1991 helposti ymmärrettäväksi ohjelmointikieleksi. Python on selkeää ja luettavaa. Koodia luetaan järjestyksessä, yksi rivi kerrallaan. Se on monikäyttöinen: se toimii monella applikaatiolla, kuten Revit, Autocad, 3ds Max, Rhino. Python sopii hyvin ensimmäiseksi kieleksi. Pythonia voi kuitenkin laajentaa C#:lla tai C/C++:lla.

Visual Basic, VB (.NET)

Kehitetty kevyeksi korkea tasoiksi ohjelmointikieleksi Microsoftin alustoille. Visual Basic. Net on toinen kieli, mutta perustuu Visual Basicin pohjalle. Basic viittaa siihen, että kyseessä on aloittelijoille suunnattu kieli. Visual Basiciä voidaan käyttää esimerkiksi Photoshopissa ja Rhinossa.

C#

C#-kieltä kehittäessä oli tarkoitus yhdistää Visual Basicin tuottavuus ja C++ teho. C# käännetään sen ajon aikana toiselle sen natiivikielelle ja sen tähden sitä voidaan käyttää eri alustoilla ilman erikseen kääntämistä. Se toimii .NET – luokkakirjaston kanssa.

Java

Java on yksi käytetyimmistä kielistä ja se toimii useilla käyttöjärjestelmillä. Se on "pieni kieli", jossa vähän varattuja sanoja. Java on kehitetty C++:sta vuonna 1991 lähtien. Javascript on sen sijaan yksinkertaisempi version Javasta. Javascript on käytettävissä Adoben eri sovelluksilla.

Processing

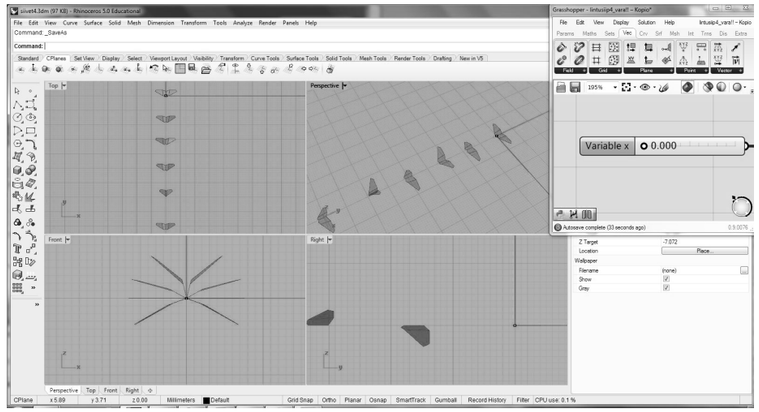
Ben Fryn ja Casey Reasin vuonna 2001 kehittämä ohjelmointikieli, jonka tarkoitus on madaltaa kynnystä tarttua ohjelmointiin. Sen yleensä katsotaan etenkin soveltuvan hyvin taiteilijoiden, graafisten suunnittelijoiden ja designerien ja niin ikään arkkitehtien käyttöön. Se sopii hyvin ensimmäiseksi ohjelmointikieleksi. Processing perustuu Javaan.

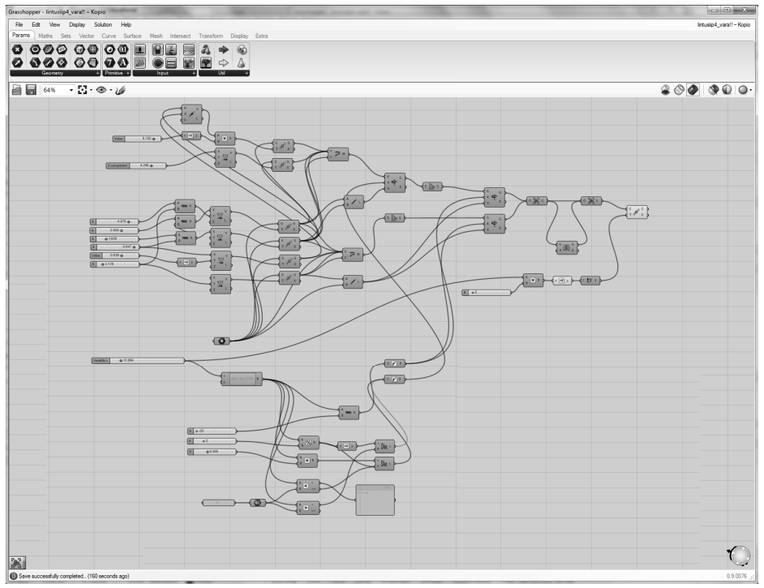
Grasshopper

Grasshopper ei ole ohjelmointikieli, mutta siihen joskus viitataan visuaalisena ohjelmointikielenä. Graafinen algoritmieditori, joka toimii Rhino3d liitoksena ja se voidaan yhdistää Rhinoscript - ja Visual Basic - kieliin. Grasshopper ei vaadi ohjelmointitaitoja.

Muita ohjelmointikieliä:

C ,Html, Mathematica, MaxScript, Maya (Maya Embedded Language (MEL) ja Python), PHP, RhinoScript (VB, Grasshopper (VB), Python), VBA



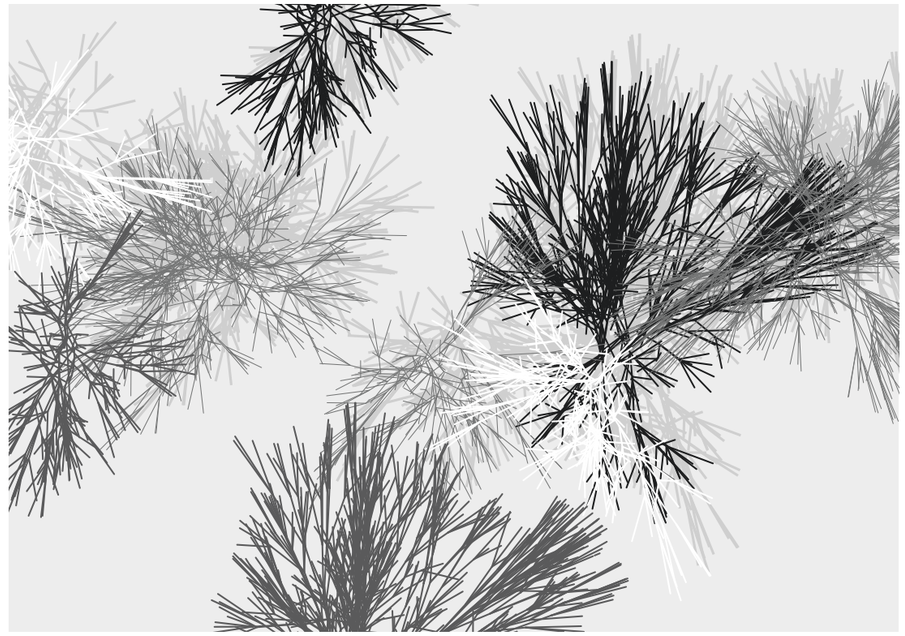


Kuvat 22., 23.
Ohjelmointiharjoituksia 1.
Tämä on ensimmäinen harjoitus josta tämä diplomotyö oikeastaan sai alkunsa. Grasshopperin harjoittelu täytyi aloittaa jostain ja päätin ensimmäisenä harjoituksena yrittää jäljitellä pihamaamme lintujen lentoa. Tämän harjoituksen lopputuloksena en päässyt haluamaani. Alkuperäisen tarkoituksenani oli automatisoida lintuparvi, jossa jokaisella linnulla olisi omat siiven iskunsa. Onnistuin vain luomaan yhden linnun. Lintua kopiodessa Grasshopperissa, malli ei enää toiminut haluamallani tavalla. Useiden yritysten jälkeen päätin siirtyä koodaamisen pariin.

Nyt paremmin Grasshopperia ymmärtävänä luulen tämä ongelma ratkeaa pelkällä Grasshopperilla, jos lintujen liikkeitä ei sen enempää määritellä. Ohjelmointi tulee peliin vain, jos haluan jäljilletellä lintuparvea, jossa sen jäsenten liike vaikuttaa toisten lintujen liikkeeseen. Ohjelmoimalla voin myös hiukan selkeyttää ja keventää Grasshopper kaaviota.

TUTKIELMIEN TYÖKALUT

Ohjelmoinnin aloittamiseksi tarvitaan ohjelmointiympäristö tai tekstieditori sekä kääntäjä. Ohjelmointi kannattaa aloittaa pienistä yksinkertaisista harjoituksista. Aion seuraavaksi esitellä hajanaisesti pienen otannan Python-ohjelmointikielen perustoimintoja, joita käytin diplomityön kolmannen osan harjoituksissa. Käsittelen kieltä toisinaan vertaillen sitä Grasshopperiin. Pelkästään lukeminen ei ole kuitenkaan hyvä tapa opetella ohjelmointia, koska eri toimintoja on todella paljon. Lisäksi pienillä variaatioilla voi saada aikaiseksi isoja muutoksia skriptiin. Yritän tässä tekstissä keskittyä suuriin linjoihin ja olla menemättä detaljeihin. Pythonin opiskeluun löytyy kyllä kattavaa kirjallisuutta sitä kaipaavalle: Parhaimmillaan opukset ovat yli tuhatsivuisia. Internetistä löytyvästä videomateriaalista pääsee kuitenkin parhaiten käsiksi aiheeseen. Kysymyksiä ohjelmointiin liittyen voi esittää erinäisille keskustelupalstoille netissä (esim. Stackexchange, Stackoverflow, grasshopper3d.com).



Kuva 24,
Ohjelmointiharjoituksia 2.
Ohjelmoimalla tehtyi puu. Puukoodi on kopioitu internetistä. Kopioinnissa on kuitenkin vaaransa. "Skriptauksesta on alkanut tulla tyyli", kuten Michael Szivos toteaa "Scripting Cultures"-kirjassa. Sen tuloksena kaikki käyttävät samoja tekniikkoja. Mitä tunnetummiksi menetelmät tulevat sitä epäkiinnostavimmaksi ne edelleen muuttuvat. (Mark Burry, 2011)

Python-ohjelmoinnin perustoiminnot

Pythonia luetaan suomen kielen kaltaisesti vasemmalta oikealla ja ylhäältä alas. Käytännössä tämä tarkoittaa sitä, että toimintoja suoritetaan rivi riviltä eikä tekstiä käännetä kerralla yhtenä kokonaisuutena. Riviä ei saa aloittaa isolla kirjaimella eivätkä ä ja ö ole käytettävissä.

Ohjelmoinnin aloittaminen:

```
>>> text = ”hello world!”
>>> print text
hello world!
>>>
```

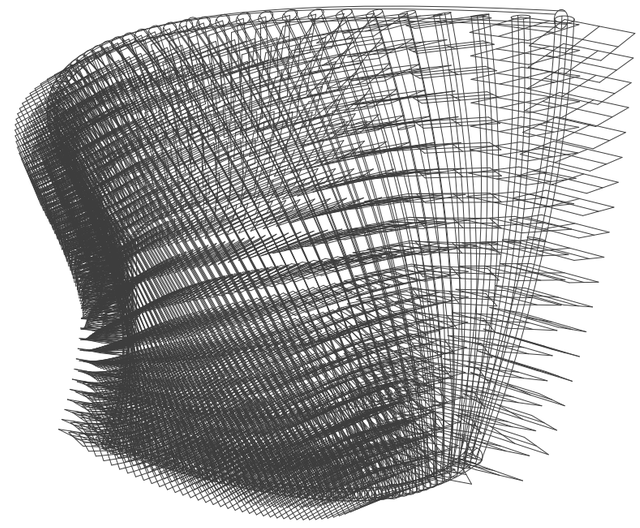
Text on **muuttuja** (variable). Se muistuttaa Grasshopperin ensimmäisen välilehden komponentteja. Pythonissa muuttujat voivat olla kuitenkin mitä tahansa. Muuttujat säilövät tietoa. Vasemmalla puolella muuttujaa on muuttujan nimi ja oikealla sen arvo. Muuttuja on syöte (input) ja arvo on tuloste (output). Lainausmerkit kertovat, että kyse on tekstistä. Print-käsky tulostaa ohjelman.

Lainausmerkeillä merkitty teksti on muuttujan tyyppi eli merkkijono (string). Muita muuttujien tyyppejä ovat esimerkiksi kokonaisluku (intVar), liukuluku (floatVar) ja totuusarvo (boolean-Var). Muuttujat nimetään aina mahdollisimman kuvailevasti ja informatiivisesti. Koodia on helpompi lukea, kun siinä ei käytetä epämääräisiä muuttujia kuten esimerkiksi x, a, tai b. Koodia kirjoittaessa on suositeltavaa olla muutenkin johdonmukainen: Muuttujat nimetään aina saman periaatteen mukaan. Kaunista koodia on helpompi ja mukavampi lukea.

Kommentteja, jotka eivät vaikuta itse koodiin, voidaan lisätä #-merkin taakse. Koodeja on voi olla vaikea ymmärtää jopa koodin kirjoittajan, jos koodin kirjoittamisesta on jo kulunut aikaa ja sen yksityiskohdat ovat jo päässeet unohtumaan. Kommentit selittävät skriptiä, jos skriptiä joudutaan myöhemmin muokkaamaan tai käyttämään jossain toisessa yhteydessä.

Lista muuttujia tehdään hakasulkeilla [...]. Listat toimivat ohjelmien tietovarastoina. Listan arvoihin voidaan viitata niiden järjestysnumeron mukaisesti listan ensimmäisen arvon ollessa 0. Esimerkiksi lista[1] tarkoittaa listan toista arvoa. Grasshopperilla geometriaa rakennetaan niin ikään listoilla, jotka saadaan Grasshopperissa näkyviin tulosteeseen (output) liitetyllä paneelikomponentilla. Paneelikomponentin sisältö vastaa Pythonissa hakasulkeiden sisältämää listaa tai listoja. Listojen muokkaamiseen Grasshopperissa on erillisiä komponentteja kuten esimerkiksi item, replace items, reverse list. jne. Listat ovat niin ikään muokattavissa Pythonilla. Listasta voidaan esimerkiksi jättää osa arvoista tulostumatta item-komponentin kaltaisesti valitsemalla tulostuvat arvot seuraavalla tavalla.

```
>>> listVar = [ 18, 26, 39, 4 ]
>>> numerot = listVar[:2]
>>> print numerot
[18, 26]
>>>
```

Kuva 25.
Ohjelmointiharjoituksia 3.
Skriptaamalla tehty kolmen käyrän orgaaninen geometria, jossa ristikkoon on lisätty panelointi.

Listat ovat tärkeä elementti rakennettaessa pisteisiin perustuvaa geometriaa. Pistegeometriaa hallitaan järjestelemällä ja hallinnoimalla listoja. Listat voivat olla sisäkkäisiä (nested list) ja ne voivat sisältää useita erilaisia listauksia.

```
>>>kirjaimet=[[”A”,”B”,”C”],[”H”,”I”,”J”],
[”M”,”N”,”O”]]
>>>harjoitus = kirjaimet[1][0]
>>>print harjoitus
H
>>>
```

Grasshopper ei pysty käsittelemään kuin edellisen kaltaista kaksi sisäkkäistä listaa sisältävän tuloksen. Kolme sisäkkäistä listaa kuten esimerkiksi [[”A”,”B”,”C”],[”H”[”I”,”J”]] vaatii Grasshopperissa listan purun vähintään kaksinkertaiseksi.

Listoja voidaan muokata useilla tavoin. Pysyvästi uusia arvoja listaan lisätään .append funktiolla.

```
>>>listText = [ ”Aino”, ”Bertta”, ”Joonas”, ”Maiju” ]
>>>listText.append(”Pyry”)
>>>values = listText [4]
>>> print values
Pyry
>>>
```

.append on python kielessä esimerkki funktiosta. Monet funktioista on Pythoniin sisäänrakennettuja kuten append, mutta funktioita voidaan lisätä ohjelmaan myös kirjastojen kautta. .Append on yksi tavallisimpia listojen jäsenfunktiota. Muita listojen jäsenfunktioita ovat esimerkiksi .remove, .count, .index, .sort .reverse jne. Grasshopperista löytyy vastaavat komponentit edellä mainittuihin toimintoihin. Funktiot liitetään muuttujiin aina pisteen avulla .append toiminnon tavoin.

Funktiokutsuja voidaan tehdä itse **def**-käskyllä, jossa sille annetaan **parametrejä** ja luodaan toimintoja. **Funktioita** usein liitetään osaksi luokkia (ks. olio-ohjelmointi).

```
>>>def lista (piste):
>>>>>>  a.append(startPoint)
>>>>>>  print piste

>>>startPoint = 1
>>>a = [5,4]
>>>lista (a)
[5, 4, 1]
>>>
```

Ehtolauseilla voidaan luoda sääntöjä, milloin esimerkiksi jokin toiminta toteutuu, kun muuntuja toteuttaa jonkin halutun arvon. Ehtolauseiden rakenteita ovat if, if not, elif ja else. Ehtolauseita voidaan yhdistää sanoilla and ja or. Näillä operaattoreilla voidaan tarkistaa, toimiiko ehto yhtä aikaa tai päteekö vain toinen ehdoista. Python lukee ehtolauseet kronologisessa järjestyksessä. Ehtolauseet muistuttavat Grasshopperin komponentteja dispatch tai cull.

```
>>>x = 3
>>>y = 4
>>>z = 4

>>>if  x <= y <= z :
>>>>>>print (”y on pienempi kuin 8, mutta suurempi kuin 1. ”)
>>>else:
>>>print (”y on jotain ihan muuta”)
y on pienempi kuin 8, mutta suurempi kuin 1.
>>>
```

while - käsky toistaa lohkonsa koodia niin kauan kuin jokin määrätty ehto on voimassa. While on näin ollen iterointia Python-kielessä. Suorituskertojen määriä ei tarvi tietää while-silmukassa ennakkoon. Suoritus päättyy, kun **silmukka** saavuttaa halutun arvonsa. While- käskyä käyttäessä tulee kuitenkin välttää loputtomia silmukoita, jotka johtavat lopulta ohjelman kaatumiseen. Silmukan muuttujan täytyy päivittyä siten, että while-ehto ei jossain vaiheessa enää toteudu. While toimii if- lausekkeen kaltaisesti.


```
>>>x = 3
>>>while x < 6:
>>>>>>print x
>>>>>>x += 1
3
4
5
>>>
```

Toinen keino silmukointiin on for i in - rakenne. Tällöin toistojen määrä tiedetään jo iteroinnin alussa. Toistoille voidaan antaa lukumäärä tai lukuväli. For-silmukka on aina korvattavissa while-silmukalla, sillä se on yksinkertaistettu versio while-silmukasta. For i in-silmukassa i on muuttuja, joka viittaa listan yksittäiseen arvoon. For i in range-rakenteella sen sijaan generoidaan 0:sta range:n annettuun paremetriin saakka. Esimerkiksi range(7) tarkoittaa 0, 1, ..., 6 lukusarjaa. Rangelle voidaan myös antaa lukuväli ja porrastus. Esimerkiki range(1, 9) generoituu luvuiksi 1, 2, ..., 8, ja range(1,9,3) generoituu 1 - 9 lukuvälin joka kolmannelle luvulle.

```
>>>for i in range(1, 9, 3):
>>>>>>print i
1
4
7
>>>
```

Range-funktiolla voidaan korvata listan arvot niiden sijainnilla. Yhdistämällä ”Range” ”For i in”-rakenteen kanssa voidaan listan arvot yhdistää niiden sijainnin kanssa.

```
>>>listNum = [3, 5, 6, 7]
>>>for i in range(len(listNum)):
>>>>>>print str(i) + ”:” + str(listNum[i])
0:3
1:5
2:6
3:7
>>>
```

for in – rakenteella pystytään myös muokkaamaan kaikkia listan jäseniä. Tällöin ei sulkeita in-sanan jälkeen.

```
>>>listNum = [1, 4, 5]
>>>a = [i + 10 for i in listNum]
>>>print a
[11, 14, 15]
>>>
```

Vaativampia ohjelmointirakenteita

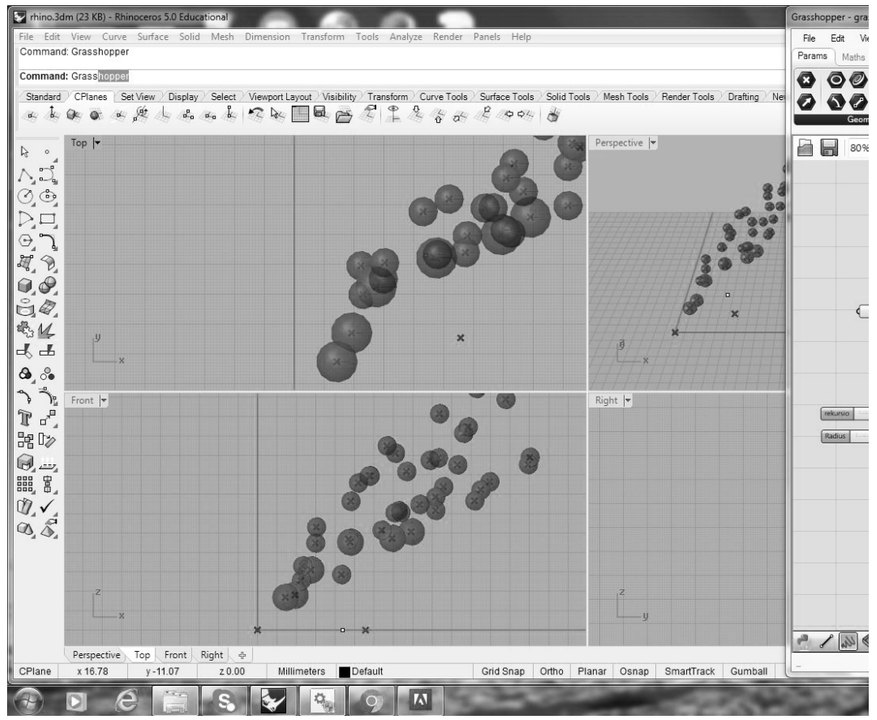
Python on olio-ohjelmointi kieli. **Oliot** ovat koodin komponentteja, josta koodi rakentuu. Oliolla on tiettyjä **ominaisuuksia** eli muutujia sekä toimintoja eli **metodeja**. Muita avain sanoja olio-ohjelmointiin ovat instanssit, perintä ja abstraktit luokat. Oliot syntyvät vasta ohjelman ajon aikana. Ne ilmentävät omaa luokkaansa, johon ne kuuluvat. Esimerkiksi luokka voi olla eläin ja eläin-luokasta voidaan luoda eri olioeläimiä, kuten koira, kissa ja lintu. Olioita voidaan käyttää uudestaan esimerkiksi jossakin toisessa ohjelmassa.

Havainnollistan oliota seuraavalla tavalla.

```
>>>class Henkilo:
>>>>>>pituus = 160
>>>>>>ika = 17
>>>>>>def vanhentuu(self):
>>>>>>>>>self.ika += 1
>>>>>>>>>self.pituus += 5
>>>>>>>>>print (str(self.ika) + ” vuotta vanha”)
>>>>>>def kasvu(self):
>>>>>>>>>if self.pituus >= 170:
>>>>>>>>>>>>print (”olen yli 170 cm pitka” )
>>>henkilo1 = Henkilo()
>>>henkilo1.vanhentuu()
>>>henkilo1.kasvu()
18vuotta vanha
19vuotta vanha
olen yli 170 cm pitka
>>>
```

self-parametria tulee käyttää aina luokkien sisällä. Luokat voivat myös periä toisiltaan ominaisuuksia. Luokissa on sisäänrakennettuja metodeja, joita erotellaan kahdella alaviivalla (_..._). Tärkein sisäänrakennettu metodi on **instanssi**. Instanssilla luodaan luokalle parametri esimerkiksi nimi, korkeus tai leveys ym.

```
>>>class Lintu:
>>>>>>  def __init__(self, laji):
>>>>>>>>>self.laji = laji
>>>a = Lintu(”pingviini”)
>>>b = Lintu(”kottarainen”)
>>>print (a.laji)
>>>print (b.laji)
pingviini
kottarainen
>>>
```

Kuva 26.
Ohjelmointiharjoituksia 6.
Rekursiivisesti kasvava pallorykelmä.

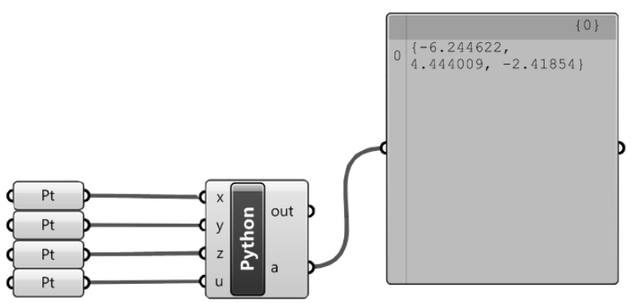
Rekursio tarkoittaa, että jokin asia määrittyy sen itsensä kautta: Funktio kutsuu itseään tietyillä ehdoilla. Rekursion avulla voidaan käsitellä suuria tietorakenteita ja tuottaa monimutkaista geometriaa ns. branching geometries. Rekursiota toimintona voidaan esimerkiksi verrata kahteen vastakkaiseen peiliin, jotka peilaavat toistensa kuvia aina loputtomiin saakka.

```
>>>def rekursioHarjoitus (num):
>>>>>>if num == 2:
>>>>>>>>>print (”valmis”)
>>>>>>else:
>>>>>>>>>print (num)
>>>>>>>>>rekursioHarjoitus(num-1)

>>>rekursioHarjoitus (5)
5
4
3
valmis
>>>
```

Python ohjelmointiympäristö Rhinossa ja Grasshopperissa

Rhino3d:n Python-ohjelmointiympäristö saadaan auki kirjoittamalla ”EditPythonScrtipt” -käsky Rhinon komentoviivalle. Grasshopperiin voidaan lisätä Python-ohjelmointiympäristö komponentin avulla. Komponentti on ladattavissa internetistä scrptit-välilehdelle. Komponentin avulla Grasshopperiin voidaan luoda omia uusia työkaluja. Python-komponentissa sen vasemman puolen muuttujat ovat komponentin syötteitä (input) ja oikealla puolella ovat komponentin tulokset (output). Syötteitä ja tuloksia voi lisätä ja poistaa tarpeen mukaan.

Python ei itsessään sisällä Rhinon toimintoja, joten koodiin tulee Grasshopperin yhteydessä lisätä **import**-käskyllä kirjastoja. Yleensä Grasshopperin kirjoitettava koodi aina aloitetaan lisäämällä Rhinoscriptsyntax-kirjasto, sillä Rhinoscriptsyntax lisää mahdollisuuden käyttää Rhinoscriptin syntaksia ja komentoja. Rhinoscript sisältää paljon hyödyllisiä funktioita. (Toni Österlund, 2017)



Kuva 27.
Python-komponentti Grrasshopperissa

Komponentin oikealla puolella olevat pisteet ovat x, y, z ja u - syötteet Python-komponenttiin. Pisteet tulee muuntaa Point3d-muotoon, joko komponentista tai rhinoscriptsyntax-kirjaston rs.coerce3dpoint(x)-toiminnolla tai rhino.geometry-kirjaston Point3d(x, y, z)-toiminnolla. Grasshopperin tulee aina tietää minkälaisesta syötteestä on kyse. Tuloste on Python-komponentista on a, joka on yllä olevassa esimerkissä pistesyöteistä muodostettu lista. Pisteistä saadaan muodostettua viivoja rs.AddCurve(points1)-käskyllä ja pintoja rs.AddLoftSrf(points2)-käskyllä. Python-komponentin koodi on seuraava:

```
import rhinoscriptsyntax as rs

point1 = x
point2 = y
point3 = z
point4 = u
a = [point1, point2, point3, point4]

print a
```

LÄHTEET

Kirjallisuus - ja verkkojulkaisu lähteet

Achim Menges, Sean Alhquist (edit.): Computational Design Thinking. (2011). John Wiley & Sons. ISBN 978-0-470-66570-1(hb) s.12, s. 68

Akos Moravanszky, Torsten Lange (Eds.): Re-Framing Identities, Architecture's Turn to History, 1970-1990 (2017) Birkhäuser Verlag GmbH, Basel. ISBN 978-3-0356-0815-1 (pdf), s. 149 – 163

Andrew Saunders: Baroque Parameters. Pablo Lorenzo-Eiroa:, Aaron Specher (edit.): Architecture in Formation. (2013) Routledge ISBN 978-0-415-53490-1 s.224

Arttu Ojanperä: Topologia. Informaatiotieteiden yksikkö, Tampereen yliopisto. (2013) [verkkodokumentti] Saatavissa: https://people.uta.fi/~eero.hyry/t/luentomuistiinpanot/Topologia.pdf (Luettu 10.01.2018)

Athina Theodoropoulou: Architectural Authorship in Generative Design. (2007) The Bartlett School of Graduate Studies, University College London. [verkkodokumentti] Saatavissa: http://discovery.ucl.ac.uk/8005/1/8005.pdf s. 16-20, 43-48

Bengt Holmström, Kultarannan päätöskeskustelu, ma 12.06.2017

Casey Reas: Process/Drawing. Helen Castle, Mike Silver (edit.) Programmin Cultures: Art and Architecture in the Age of Software. (2006) Architectural Design ISBN-13 9780470025857 s. 26 - 33

Christopher Alexander: A Pattern Design.(1997) Oxford University Press ISBN 0-19-501919-9

C.Thomas Mitchell: Redefining Designing, From Form to Experience (1993). Van Nostrand Reinhold. ISBN 0-442-00987-9 s. 52

David Cohn: Evolution of Computer-Aided Design. [verkkodokumentti] Saatavissa: http://www.digitaleng.news/de/evolution-of-computer-aided-design/ (Luettu 25.06.2017)

David Jason Gerber: Parametric Tendencies and Design Agencies. Dr. david Jason Gerber, Mariana Ibanez (edit.): Paradigms in Computing, Making, Machines, and Models for Design Agency in Architecture. (2014) eVolo Press. IBNS 978-1-938740-09-1 (hardback) s. 387-399

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software (1997). Addison Wesley Longman, Inc. ISBN 0-201-63361-2 s. 1-4

Frederick P. Brooks: No-Silver-Bullet - Essence and Accident in Software Engineering. (1986). [verkkodokumentti] http://www.itu.dk/people/hesj/BSUP/artikler/no-silver-bullit.pdf

Frieder Nake (2017). "To ask a permission to use picture in an architecture thesis in Oulu University" Sähköpostiviesti 12.01.2017. Vastaanottaja Maija Poukka.

George L Legendre, Helen Castle (edit.): Mathematics of Space (2011) Architectural Design ISBN 0003-8504

Gordon Pask: An Evolutionary Architecture (1995). Themes VII, Architectural Association London ISBN 1-870890-47-7 s. 7-6

Gregg Lynn: an Advanced form of Movement. Maggie Toy (Edit.): Architecture After Geometry (1997) ISNN: 003-8504 s. 54 - 55

Helen Castle: Editorial. Helen Castle (edit.): Patterns of Architecture. (2009) John Wiley & Sons. ISBN 978-0470 699591 SIVUNUMERO s.5

John Frazer: An Evolutionary Architecture (1995). Themes VII, Architectural Association London ISBN 1-870890-47-7 s. 7-6

Jules Moloney: Designing Kinetics for Architectural Facades (2011) Routledge ISBN 978-0-415-61033-9 (hbk) s. Cybernetics and Architecture

Kostas Terzidis: Algorithmic Form. (2003). Achim Menges, Sean Alhquist (edit.): Computational Design Thinking. (2011). John Wiley & Sons. ISBN 978-0-470-66570-1(hb) s.12, s. 94

Kristoffer Josefsson: Symmetry as Geometry Kuwait International Airport. Helen Castle, Brady Peters, Xavier de Kestelier (edit.) Computation Works, the Building of Algorithmic Thought. (2013) ISSN: 003-8504)

Lisa Iwamoto: Digital Fabrication, Architectural and Material Techniques.(2009) Princeton Architectural Press, New York. ISBN 978-1-56898-790-3 s. 4-5

Malcolm McCullough: 20 Years of Scripted Space. Helen Castle, Mike Silver (edit.) Programmin Cultures: Art and Architecture in the Age of Software. (2008) Architectural Design ISBN-13 9780470025857 s.12-15

Manuel Kretzer: Information Materials, Smart Materials for Adaptive Architecture (2017) Springer ISBN 978-3-319-35148-3 s. 11

Marjan Colletti: Exuberance and Digital Virtuosity. Marjan Colletti. Helen Castle (edit.): Exuberance . (2010) ISSN: 0003-8504 s. 8

Mark Burry : Scripting Cultures, Architectural design and programming (2011). John Wiley & Sons Ltd. ISBN 978-0-470-74641-7 (Paperback)

Micheal Hansmeyer: the Sixth Order. Pablo Lorenzo-Eiroa:, Aaron Specher (edit.): Architecture in Formation. (2013) Routledge ISBN 978-0-415-53490-1 s.212

Michael Meredith: Before and After Geometry. Helen Castle, Brady Peters, Xavier de Kestelier (edit.) Computation Works, the Building of Algorithmic Thought. (2013) ISSN: 003-8504 s. 98

Mike Silver: Pattern Deposition from Scripts to Apllications. Helen Castle (edit.): Patterns of Architecture. (2009) John Wiley & Sons. Architectural Design. ISBN 978-0470 699591 s. 94-99

Pablo Lorenzo-Eiroa: Form: in: Form on the Relationship between digital Signifiers and Formal Autonomy. Pablo Lorenzo-Eiroa:, Aaron Specher (edit.): Architecture in Formation. (2013) Routledge ISBN 978-0-415-53490-1 s.18-19

Pablo Miranda Carranza, Annie Locke Scherer. (2017). To ask help for thesis about "scripting as method in architecture" Sähköpostiviesti 1.10.2017. Vastaanottaja Maija Poukka. (ks. liite s.68)

Patrik Schumacher: Parametricism as Style. (2008) [verkkodokumentti] http://www.patrikschumacher.com/Texts/Parametricism%20as%20Style.htm (Luettu 12.01.2018)

Robin Evans: Translation from drawing to Building. (1997) [verkkodokumentti] Saatavissa: http://arts.berkeley.edu/wp-content/uploads/2016/01/arc-of-life-Robin_Evans_Translations_From_Drawing_to_Building1.pdf (Luettu 03.1.2018)

Toni Österlund: MaijaPoukka_Diplomityö_kommentoitu.pdf Vastaanotettu dokumentti: 24.01.2018. Vastaanottaja Maija Poukka.

William J. Michell: A New Agenda for Computer-Aided Design. Computtational Design Thinking. (2011). John Wiley & Sons. ISBN 978-0-470-66570-1(hb) s. 92. [Alkuperäinen lähde] The Electronic Design Studio: Architecturel Education in the Computer Era. (1990) MIT Press.

Kuvalähteet:

Kuvat tekijän ellei toisin mainittu. Työssä oleviin kuviin on pyydetty luvat tekijänoikeuksien omistajilta.

Kuva 1. :
Ivan_Sutherland1962. Saatavissa: http://history-computer.com/ModernComputer/Software/Sketchpad.html (Avattu 3.8.2017)

Kuva 2. :
6/7/64 Nr. 20 Zufälliger Polygonzug. Saatavissa: http://dada.compart-bremen.de/item/artwork/428 (Avattu 1.8.2017)

Kuva 3. :
Nilas Sculpture: Eero Lunden, Markus Wikar, Toni Österlund. Saatavissa: http://www.osterlund-ark.fi/archives/portfolio/nilas/ (Avattu 18.1.2018)

Kuva 4.
ShinMinamata Station. Saatavissa: http://www.makoto-architect.com/three_st/image/in01e.htm (Avattu 3.7.2017)

Kuva 5.
ShinMinamata Station (Kuvan nimi puuttuu.) http://www.makoto-architect.com/three_st/image/DSCN0112e.htm (Avattu 3.7.2017)

Kuva 6.
One Ocean, Thematic Pavilion EXPO 2012 / soma. Kuva: Soma Otettu (5.12.2012)

Koodit:

Ohjelmointiharjoituksia 2. : Intro to Python Scripting 12 | Converting an Existing Script - Branching. [verkkovideo] Saatavissa: https://www.youtube.com/watch?-v=58iNw1qcQKs&index=12&list=PLGV167zE8gnVhurBT46afZ1RlK9RzAsLx (Katsottu 20.8.2017)
Ohjelmointiharjoituksia 4. : Python Scripting in Grasshopper - Scripting Form 1/2 [verkkovideo] Saatavissa: https://www.youtube.com/watch?v=sB-sc73CSwI (Katsottu 20.8.2017)

Pablo Miranda Carranza: VS: To ask help for thesis about "scripting as method in architecture" Sähköpostiviesti 1.10.2017. Vastaanottaja: Maija Poukka.

Terve Maija,
(I was an exchange student in Helsinki, long, long time ago!)

There is not much more course material to send than the one available on the site… it is the first time we run the course so we are filling it up as it goes. In the blog, under my name (teachers) there is a link to the Processing and Arduino tutorials I use in teaching, you can also reach them here: automatic.se

To answer your questions quickly:

1.Do you think scripting is an essential skill for an architect in future?
No, or at least I sincerely hope not. I believe that programming has an important role to play in architecture, but not an essential one. Programming is a way of producing qualities that are not representable or workable through other media, like drawing (digital or on paper). It is also the medium regulating an increasing part of our lives, not only the programs we use for drawing, but how we retrieve information or establish social relations (It would be impossible to communicate with you right now as I am doing if it wouldn't be for programming… from the text editing inside my email program to all the routing of information when I press the send button…). In this sense programming literacy should be something that anyone (not only architects) should have some familiarity with. But there lots of things that are central to architecture that are very hard to do through programming, and that are best done in more traditional, designerly ways. It is for example questionable that architecture as a profession and discipline could exists without drawing, and the relation between them is clear in for example the work of Robin Evans, and even Mario Carpo. Programming is interesting because one can tap into qualities that are outside the possibilities and limitations of drawing. It is only useful insofar one wants to investigate those qualities.

2.Why would you script, since you can do it - almost surely - as well in grasshopper?
I script because it is fun! but in reality grasshopper is very limited in comparison to programming (or scripting, in for example Python). The limitations of graphic programming (like Grasshopper) were already pointed out during the mid eighties by Fred Brooks, when graphic programming was a new thing https://en.wikipedia.org/wiki/No_Silver_Bullet (the wikipedia article does not explicitly mention graphic programming, but the paper does). The thing is that programming languages are what is called "Turing Complete" https://en.wikipedia.org/wiki/Turing_completeness , in layman words it means that any computer language can express any logical statement possible (if you can explain a causal process of anything like a thought, a machine, or a natural phenomenon, you can represent it as a program or a "Turing Machine"). Grasshopper is far from being Turing complete… it represents what is called the data-flow, and is not capable of representing another central concept of programs, which is called control flow https://en.wikipedia.org/wiki/Control_flow (not the best wikipedia article, but control flow is essential for imperative programming languages to be able to be "Turing complete"… functional languages, which are a bit more esoteric, have alternative concepts, like recursion, which is also impossible to do in Grasshopper). This is why people, whenever they need some real functionality, need to program it or script it (using for example Python or the C# language), and then perhaps make it available as a plug-in for Grasshopper. So as you see, "there is no silver bullet" that will do away with programming (Grasshopper, as Rhino and every other application in your computer, has been, at some point, a text written in a programming language, specifying all its functioning…). That means that programming (or scripting, for the difference check: https://stackoverflow.com/questions/17253545/scripting-language-vs-programming-language#17253557) is far-far more expressive than a data-flow model like that in Grasshopper. Don't misunderstand me, I think that Grasshopper is great and real fun, but programming is possibly the most important revolution in forms of writing since the invention of the alphabet, as media theorist Friedrich Kittler, among others, have pointed out: http://www.ctheory.net/articles.aspx?id=74, and I think it is worth considering how this affects architecture.

Long answers, I hope that it is of some use to you!

Pablo Miranda Carranza
pablo.miranda@arch.kth.se
KTH School of Architecture
100 44 Stockholm, Sweden

KIITOS / THANK YOU,

vuosikurssi 11, opintovuosista ja siitä, että olen saanut kasvaa arkkitehdiksi teidän kanssanne,
Rainer Mahlamäki ja Aulikki Herneoja kannustuksesta ja ohjauksesta,
Toni Österlund ja Tuulikki Tanska.

Kiitos Toni, kahden tunnin diplariohjauksesta 23.10.2017. Annoit parhaat lukuvinkit, kritiikin ja opin.
Tuulikki, kiitos diplarin alkuun laittamisessa, neuvoista, tuesta ja ymmärryksestä. Tiedät, että kärsivällisyys on koetuksella, kun koodin kääntäjään ilmestyy teksti “SyntaxError“.

ありがとう dear Akane Imai, for taking me to the Digital Fabrication labroratory of University of Tokyo and Your friendship and advicies,
Pablo Miranda Carranza and Annie Locke Scherer from KTH, your messages made me believe in the topic on half way when I had hard time,
ja äiti, tekstin tarkastuksesta ja korjauksista.